COLOMBO SCIENTIFIC PUBLISHING

INTERNATIONAL JOURNAL OF COMPUTER SCIENCE & INFORMATION SYSTEM
E-ISSN: 2536-7919
P-ISSN: 2536-7900
PAGE NO: 09-19

Server-Side Profiling As A Methodology For Proactive Identification Of Performance Bottlenecks

Ilia Detinkin*
Independent Researcher, London, RM12 6EG, United Kingdom
*Corresponding author. E-mail: ilya.detinkin@gmail.com

ARTICLE INFO
Article history:
Submission: September 01, 2026
Accepted: September 29, 2026
Published: October 24, 2026
volume: Vol.11 Issue 10 2026
DOI: -
10.55640/ijcsis/Volume11Issue10-02

Keywords:
server-side profiling; performance bottlenecks; query plan analysis; proactive optimization; PostgreSQL; software performance engineering

ABSTRACT
Growth in stored data and request volume turns latency degradation into a recurring condition of long-lived server systems. A frequent response is to add hardware capacity, which raises operating expenditure while the underlying defect persists. This article treats server-side profiling as a proactive engineering practice and organizes it into a four-stage procedure: localization of the layer that holds the delay, selection of a diagnostic instrument according to whether the source is evident, prioritization of findings by their share of total execution time and by call frequency, and targeted remediation followed by re-measurement. The evidence comes from two production cases on a PHP and PostgreSQL platform, documented through a descriptive multiple-case design. The first case concerns an inventory listing endpoint whose response time had grown to roughly 10 seconds. Query plan analysis revealed a recursive traversal within the main statement, incomplete index coverage for join and filter predicates, a hash operation spilling to temporary files, and expensive computed columns evaluated on rows that pagination would discard. Response time fell to roughly 500 milliseconds after four targeted changes. The second case concerns bulk warehouse intake, where a call-level profiler attributed the delay to many small recalculations repeated after every saved item, and deferring those operations into a single pass at the end of the request removed the redundancy. The cases occupy opposite ends of a diagnostic spectrum covered by one procedure, and the article argues that added capacity conceals both classes of defect while leaving them in place. The evidence base is limited to two cases from a single platform, so the findings support the transferability of the procedure rather than a statistical claim about effect size. The practical implication for engineering teams is to capture baselines for critical endpoints while they still perform acceptably, so that later degradation becomes a comparison against a known quantity rather than an incident.

1. INTRODUCTION

Server systems that stay in production for several years accumulate data at a pace their original design seldom anticipated. A statement that returned in 40 milliseconds against 10,000 rows may take 4 seconds against 4,000,000 rows, and the transition occurs without a deployment, a configuration change, or any event that an incident log would record. Degradation of this kind arrives as a slope, and the slope crosses the threshold of user tolerance long after the design decision that produced it.

The response available to an engineering team falls into 2 families. Capacity can be enlarged through a bigger instance, additional replicas, or a higher service tier, and the second family locates the cause and removes it. The first option executes faster, carries little deployment risk, and requires no changes to

application code, which explains its popularity among teams under delivery pressure. Its cost accrues every month and grows along the same data curve that produced the slowdown.

Capacity also has a blind spot. A hash join that exceeds the memory budget of a single query operation writes temporary files to disk, and the penalty follows from memory configuration more than from processor count. A handler recomputing derived warehouse totals after every saved item repeats the same arithmetic thousands of times per batch, and faster hardware shortens each repetition while preserving their number. Both defects survive an upgrade and return at the next volume threshold.

Profiling is a familiar practice in engineering, and its typical mode of use is reactive. An incident opens, an engineer attaches a profiler, the offending function surfaces, a patch ships, and the instrument returns to the shelf until the next outage. A proactive posture inverts the sequence, since measurement precedes complaint, and the profile of a critical endpoint is captured while the endpoint still performs acceptably. A defect visible at 300 milliseconds costs less to remove than the same defect at 10 seconds, after it has acquired dependent code and user habits.

Research on software performance covers instruments and defect catalogs in depth, and the path between these bodies of work receives less attention. An engineer holding a profiler output faces a ranked list of functions and a plan tree with dozens of nodes, and the decision about which finding deserves an hour of work depends on judgment that published guidance rarely formalizes.

This article addresses that path. The objective is to formulate server-side profiling as a proactive methodology that connects a measured signal to a prioritized and verified remediation, and to demonstrate it in 2 production cases. Three tasks follow: organizing existing diagnostic practice into an explicit sequence of stages, applying that sequence to 2 cases from a high-load commercial platform, and extracting a transferable pattern whose validity does not depend on the runtime or the database engine used.

The contribution claimed here is the systematization. Neither the instruments nor the individual optimizations described below are new, and to the author's knowledge, no claim of priority over any of them would withstand scrutiny. What the literature appears to lack is a compact procedure that specifies how a profiling signal yields a ranked decision and how that decision is verified. The 2 cases occupy opposite ends of a diagnostic spectrum: one presents a single dominant operation, and the other an aggregate of small operations with no dominant member; a methodology handling one regime alone would stall on half the problems an engineer encounters.

1.1. Diagnostic instruments for server-side systems

Instruments for observing server behavior are divided by how they collect data. Sampling profilers interrupt execution at a fixed frequency and record the active call stack, yielding a statistical picture at predictable overhead. Instrumenting profilers wrap function entry and exit, producing exact call counts and per-function timings at a cost proportional to call volume. XHProf, used in the cases below, belongs to the second family and reports invocation counts along with inclusive and exclusive wall-clock times. The distinction matters for diagnosis, because a defect built from many cheap repetitions is visible in a call count and can hide inside the noise of a sampled stack.

Tool selection has become a research question in itself. A systematic comparison of 30 open tracing tools concludes that each option offers a distinct combination of strengths with no dominant choice across contexts [1]. Practice within large organizations has converged on pipelines that combine collection, storage, and analysis, and an industry survey of engineers from 10 companies reports that such pipelines are standard, while their implementations diverge at the analysis stage [2].

1.2. Diagnosis at the data layer

Relational engines expose their execution strategy through a plan. The PostgreSQL command EXPLAIN ANALYZE executes the statement and returns the execution plan tree, including estimated and actual row counts, per-node timing, loop counts, and memory or disk usage for sorts and hash builds. Divergence

between estimated and actual cardinality is the primary diagnostic signal, since a planner expecting 50 rows and receiving 500,000 rows chooses a nested loop join when a hash join belongs, and the plan degrades by orders of magnitude while every operation remains correct.

Automated index selection has been studied for decades and remains an open problem. An evaluation of index advisors across workloads and engines reports wide variation in recommendation quality and identifies benefit estimation as the dominant source of error [3]. Advisors query the optimizer through a what-if interface, costing hypothetical configurations without materializing them, and measurements of that interface show errors large enough to compromise the resulting recommendations [4]. A consequence follows for the methodology proposed here: an engineer who trusts a cost estimate reasons about a model of execution, whereas an engineer who runs EXPLAIN ANALYZE reasons about execution.

1.3. Recurring defects in data access code

Defects in the code that reaches a database recur with enough regularity to support catalogs. A study of data-intensive systems identifies data-access anti-patterns across relational stores, document stores, and polyglot persistence and organizes them by root cause into categories, including unnecessary computation, inefficient retrieval, and excessive data transfer [5]. Coverage beyond object-relational mapping matters here, since the platform below issues SQL directly.

Quantitative evidence about where these defects concentrate comes from an examination of 423 database access defects collected from 7 large Java projects backed by MySQL or PostgreSQL. The reported taxonomy separates 5 root cause categories, and defects located in SQL statements, in the schema, and in the access API account for 84.2% of the total [6]. An engineer beginning at the statement and the schema addresses the majority of realistic causes before examining configuration or result handling, which is the order the methodology below encodes.

1.4. Measurement as a repeated practice

Performance regression testing has received less study than functional regression testing, a gap noted in work that adapts test case prioritization to software microbenchmarks [7]. The imbalance is evident in practice, since projects that run functional tests on every commit often measure performance only when a user complains.

Linking fine-grained measurements to system behavior is the second obstacle. An evaluation of microbenchmark suites as detectors of application-level change reports that suites can identify such changes, while their sensitivity depends on the coverage of the code paths that dominate application time [8]. The methodology below treats baseline capture as a stage with its own output, since a measurement covering an unrepresentative path yields a baseline that a later comparison cannot use.

The reviewed sources describe instruments, catalog defects, and quantify where defects concentrate. The procedural link receives less treatment, and no source encountered during preparation specifies how an engineer moves from a ranked profiler output to a decision about which finding to address first and what measurement confirms the outcome. The present work occupies that gap, and nothing below proposes a new instrument or optimization.

2. Materials and Methods

2.1. Study design

The study combines a methodological synthesis with a descriptive multiple-case design covering 2 cases. The synthesis assembles established diagnostic practice into an ordered procedure, and the cases test whether that procedure describes what happened when the practice was applied to production defects. The design carries the limitations of its class, since no control group exists, and the measurements come from a live system where the background load varied.

2.2. Case selection and anonymization

Both cases come from a commercial inventory management platform built on PHP with a PostgreSQL backend. Selection followed 2 criteria. Each case had to include a measured outcome, and the pair had to cover both branches of the second stage of the procedure (Section 3.1), so that the procedure would be exercised in the evident-source and unclear-source regimes. The cases were anonymized before analysis. Table and column names were replaced with generic equivalents; no data values are reproduced; and no detail permits attribution to a specific employer or customer.

2.3. Metrics and measurement conditions

End-to-end wall-clock time of the request served as the primary metric in both cases, measured on the production configuration under normal working load. Secondary metrics differed by branch: plan analysis contributed per-node actual time, actual and estimated row counts, loop counts, and temporary file volume, while code profiling contributed per-function inclusive and exclusive time, invocation counts, and peak memory. Comparisons before and after each change kept the endpoint, filter parameters, and page position constant.

3. Results

3.1. The four-stage procedure

The procedure consists of 4 stages arranged as a cycle, shown in Fig. 1.

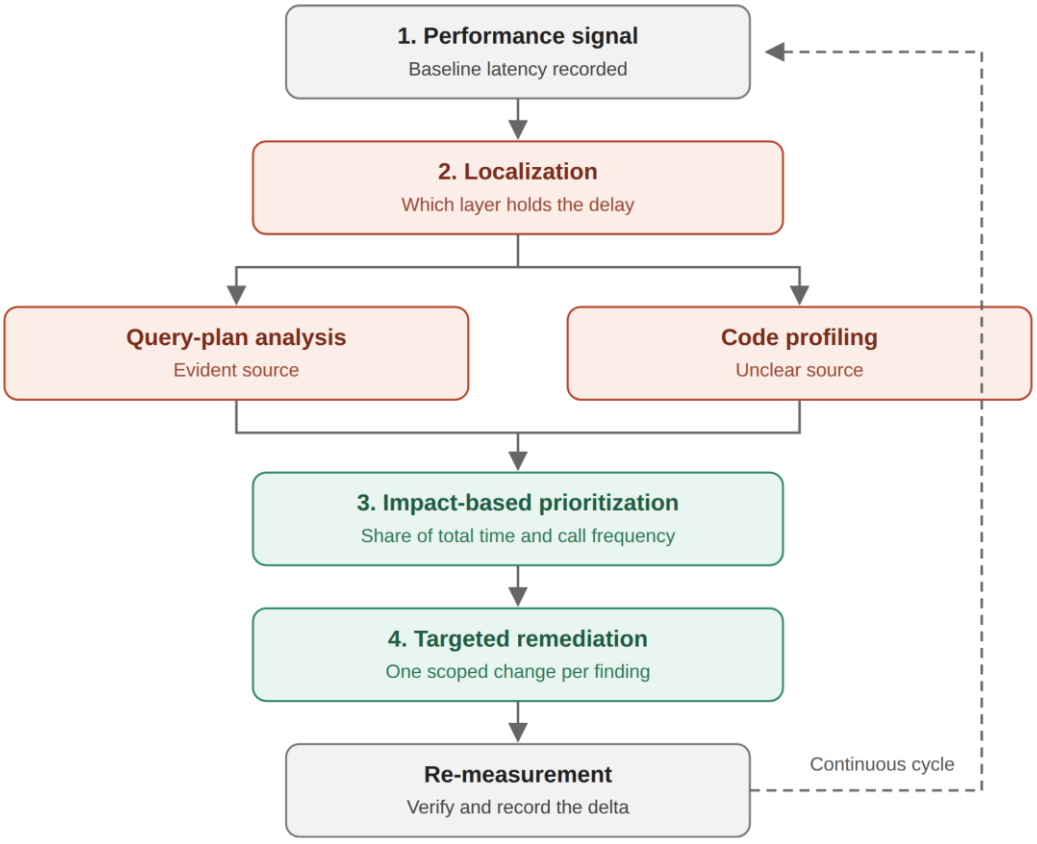


Fig. 1. Four-stage cycle of proactive server-side profiling. Stage 2 branches into query-plan analysis (evident source) or code profiling (unclear source); the dashed return path indicates that re-measurement establishes the baseline for the next iteration.

The first stage records a performance signal and fixes a baseline. The signal may arrive from a user report, a monitoring threshold, or a scheduled measurement, and its origin does not alter what the stage produces, namely, a numeric response time attached to a named operation under stated conditions. Absence of this

number makes every later claim unfalsifiable.

The second stage localizes the delay to a layer, and a branch opens according to how evident the source is. When a single operation dominates the request, the plan of that statement is the appropriate instrument, and EXPLAIN ANALYZE returns the per-node timings and cardinalities needed to proceed. When no single operation dominates, a code profiler identifies which functions consumed the request and how many times each was invoked. Choosing the wrong branch costs an iteration without corrupting the result, since a plan analysis that finds no anomalies redirects the engineer toward the correct code path.

The third stage prioritizes findings by impact, and the ranking depends on 2 quantities. The share of total request time attributable to an operation sets the ceiling for any improvement, and the frequency of invocation determines whether a small per-call cost adds up to a large total. An operation consuming 4% of request time offers at most 4% of improvement, and a function costing 200 microseconds that executes 8,000 times per request costs 1.6 seconds. Ranking by these quantities prevents the failure mode in which an engineer optimizes the most conspicuous code path rather than the most expensive one.

The fourth stage applies a targeted change and measures again. Scope discipline governs it, since a change addressing a single ranked finding permits attribution of the resulting delta, whereas a batch of simultaneous changes forfeits that attribution. The re-measurement uses the conditions fixed in the first stage, and its output serves as the new baseline. The dashed return path in Fig. 1 encodes the proactive character of the procedure, because the cycle closes into a practice with no terminal state.

3.2. Case A: inventory listing endpoint

The platform served a page listing stock held in the company warehouse and in supplier warehouses. Data reached the page via a separate AJAX request that returned JSON, and after several years of data growth, the response time grew to roughly 10 seconds. The endpoint executed a single large SQL statement that included filters, sorting, pagination, and several computed columns, which placed the case in the evident-source branch.

One element of the statement deserved attention before the plan was read. Interchangeability between part numbers formed a directed and transitive relation, so that a part replacing a second part, which in turn replaced a third, also replaced the third, while the reverse direction carried no guarantee. A recursive subquery walked this relation inside the main statement alongside every filter and computed column.

Analysis through EXPLAIN ANALYZE produced 4 findings, which Table 1 summarizes together with the change applied to each.

Table 1. Diagnostic findings in Case A and the corresponding changes

Diagnostic signal	Diagnosed cause	Applied change
Recursive node dominating plan runtime	Transitive substitution walk executed inside the main statement	Walk extracted; matching identifiers materialized into an indexed temporary table
Sequential scans on join and filter predicates	Index coverage no longer matched the grown schema and volume	Audit of predicates followed by creation of the missing indexes
Hash operation writing temporary files	Working memory for one operation exceeded after data growth	work_mem[a] raised for the session executing the statement and reset afterward
Computed columns evaluated for discarded rows	Subqueries and functions applied before pagination truncated the result	Statement split into two levels, with the inner level selecting the page

[a] work_mem: PostgreSQL parameter bounding the memory available to a single query operation, such as a

sort or a hash build.

The first change removed the recursive walk from the main statement. Acceptable nomenclature identifiers were collected separately, written into an indexed temporary table, and then filtered by the inventory query, allowing the planner to cost 2 simpler problems in place of one compound problem. The second change followed an audit of join and filter predicates, which showed that index coverage had not kept pace with schema growth, and the missing indexes were created.

The third change addressed a configuration parameter whose effect stayed invisible until the data crossed a threshold. The PostgreSQL setting work_mem bounds the memory available to an individual query operation, such as a sort or a hash build, and an operation exceeding the bound spills to temporary files on disk. One hash build had grown past the configured limit, and the resulting disk activity accounted for a large share of the runtime. Since the underlying data was not expected to grow further, the limit was raised via SET SESSION work_mem for the session executing the statement and released via RESET work_mem afterward, leaving every other connection unaffected.

The fourth change concerned pagination. A clause of the form LIMIT 100 OFFSET 1000 requires the engine to locate at least 1,100 qualifying rows, discard the first 1,000, and return the remainder. Expensive computed columns in the select list, including a correlated aggregate and a per-row function call, were evaluated across all rows the engine touched before the offset removed most of them. Restructuring placed filtering, ordering, and pagination in an inner query returning identifiers alone, and the outer query computed the expensive columns for the 100 surviving rows. Fig. 2 contrasts the arrangements.

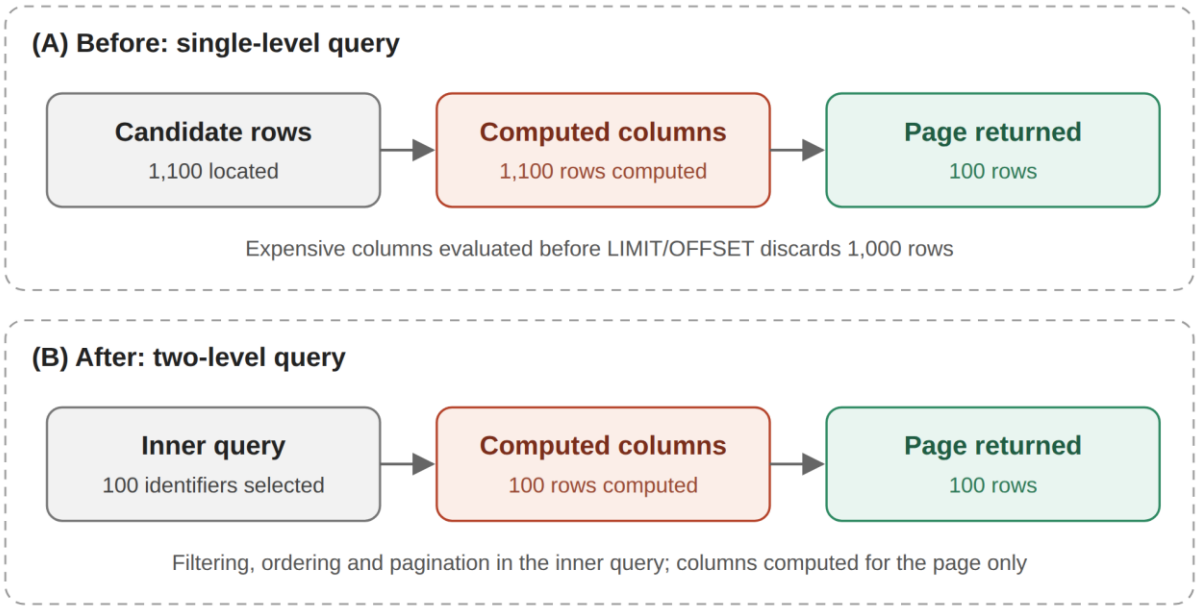


Fig. 2. Placement of computed columns under single-level and two-level pagination for a page of 100 rows at offset 1,000. (A) Single-level query: expensive columns are evaluated for all 1,100 candidate rows before LIMIT/OFFSET discards 1,000 of them. (B) Two-level query: the inner query performs filtering, ordering, and pagination and returns 100 identifiers; the outer query evaluates the expensive columns for those rows only.

The restructuring left the offset cost itself untouched, since the engine still had to locate and skip the preceding rows. Product requirements permitted no alternative, as users could open any page directly and sort by any column, thereby excluding keyset pagination. The change reduced the number of rows subject to expensive per-row computation by a factor of 11 at the page position shown, and this factor increases with the offset.

Response time for the endpoint fell from roughly 10 seconds to roughly 500 milliseconds after all 4 changes, a reduction by a factor of about 20. The changes were applied sequentially, with measurements between

them, and the largest single contribution came from removing the recursive walk. Precise per-change attribution is withheld because the interventions interact.

3.3. Case B: bulk warehouse intake

The second defect surfaced when the company began receiving larger consignments. Adding a consignment to the warehouse grew slow enough to disrupt the work of warehouse staff, and no single operation stood out as a candidate, which placed the case in the unclear-source branch.

XHProf was attached to the AJAX request that performed the addition, and the resulting profile was ranked by inclusive time and by invocation count. Saving an individual item proved fast in isolation, and the cost came from auxiliary handlers that ran after each saved item to refresh derived warehouse figures. Each execution took little time, and a large consignment triggered the same recalculations once per item, so the aggregate dominated the request, while no individual entry in the profile appeared anomalous.

Remediation changed the batch's control flow. Requests for derived recalculations were accumulated during processing and executed once at the end of the request, eliminating repetition while preserving the final state of the derived data. Warehouse staff observed a reduction in intake time, and no instrumented measurements of the before-and-after states were retained, which the limitations below address. The difficulty here lay in the diagnosis, since the fix required a modest change to the order of operations.

3.4. Comparison across the 2 cases

Table 2 places the cases side by side along the dimensions that the methodology distinguishes.

Table 2. Two diagnostic regimes covered by a single procedure

Dimension	Case A: inventory listing	Case B: bulk intake
Detection regime	Evident source, one dominant statement	Unclear source, no dominant function
Instrument	Query plan analysis through EXPLAIN ANALYZE	Call-level code profiler XHProf
Defect class	Single high-cost operation	Many low-cost operations repeated per item
Decisive metric	Node runtime, row counts, temporary file activity	Invocation count multiplied by inclusive time
Remediation class	Restructured data access and statement shape	Deferred and batched redundant recomputation
Measured outcome	About 10 s reduced to about 500 ms	Reduction without retained instrumentation

The rows describing detection and defect class carry the analytical weight of the comparison. Case A yields to an instrument reporting where time was spent, because time concentrated in one place, while Case B defeats that instrument and yields to one reporting how often something happened, because the total was assembled from individually unremarkable parts. A practice equipped for one regime alone resolves the first class of defect and stalls on the second.

4. Discussion

Abstracted from their details, both cases traverse the same 3 steps. A measured signal identifies where cost accumulates, a ranking by share of total time and by frequency selects among competing findings, and a change scoped to one finding is verified against the baseline. The abstraction survives differences in instruments because plan node timings and profiler call counts answer the same question in different units, and both materializing an intermediate result and deferring a repeated computation reduce the work

performed per unit of delivered output.

Reading the cases as opposite regimes explains a common failure in practice. An engineer who has resolved several first-class defects expects profiling to reveal a culprit, but that expectation misfires with a second-class defect, where the profile shows a flat distribution across many entries. Ranking by invocation count converts that distribution into an ordered one, which is what allowed Case B to proceed.

A procedure that runs only during incidents produces baselines nobody recorded. Profiling a critical endpoint before release turns the profile into an artifact reviewed alongside the code, and a reviewer who sees a function invoked 8,000 times per request can question the design while the change remains cheap to alter. Performance budgets attached to endpoints turn degradation into a threshold crossing detected by automation, and periodic re-measurement against fixed baselines detects the slope described in the introduction.

Monitoring the stability of query plans deserves separate mention, because the failure it detects has no code change behind it. A planner selects a strategy from statistics, statistics track a growing table, and the strategy changes when an estimate crosses a boundary in the cost model. The application ships no release, the schema receives no migration, and the endpoint slows down, a mechanism Case A exhibits in the hash operation that began spilling to disk.

Neither defect examined here would have been removed by a larger server. The hash operation in Case A spilled to disk because one query operation exceeded its memory allowance, and additional processor cores addressed neither the allowance nor the resulting disk activity. More memory postpones the spill until the data grows again, which converts a defect into a schedule. The repeated recalculations in Case B execute the same number of times on faster hardware, so that the total shrinks by the hardware ratio while the redundancy survives.

The economic asymmetry between the 2 responses compounds over the life of a system, since capacity is priced per unit of time and scales with the workload that an unremoved defect generates, whereas a code change is priced once in engineering hours. Evidence for the second option comes from an industry case study reporting reduced consumption achieved through refactoring alone, with the infrastructure held constant [9]. Capacity remains the correct response to genuine demand growth, and the argument advanced here concerns the case where a measurable defect exists and capacity is purchased in place of diagnosis.

PHP and PostgreSQL functioned as the setting of the cases without constituting their subject, and each element of the procedure has counterparts elsewhere. Call-level profiling with invocation counts is available via async-profiler on the JVM, py-spy for Python, and pprof for Go, and plan inspection is available in every mature relational engine. The prioritization rule references no product, since the share of total time and frequency of invocation are properties of any measured execution.

One boundary deserves a statement. The cases concern a monolithic application in which the layer boundary between code and storage resides within a single process. Distributed deployments spread the same question across service boundaries and require trace correlation; dedicated methods for multi-dimensional root cause localization exist in that setting [10]. The 4 stages appear to transfer, and their instruments change.

The evidence base consists of 2 cases from a single platform in a single domain, which does not support a statistical claim about frequency or effect size across server systems in general. Measurements were taken under variable background load, and no controlled environment isolated the interventions from workload variation.

Attribution within Case A is incomplete. The 4 changes were applied in sequence and interact, so the aggregate reduction cannot be decomposed into 4 independent contributions, and a controlled replication applying each change in isolation against a fixed dataset would supply that decomposition.

Case B carries a heavier limitation, since its outcome is reported qualitatively. No numeric before-and-after

comparison was retained, so the claim of improvement rests on the mechanism and on operator observation, and a reader may weigh it below the quantified outcome of Case A. Both cases were documented after the fact by the engineer who performed the work, introducing the recall and confirmation risks associated with retrospective reports.

Further work follows 3 directions. A controlled replication of Case A on a synthetic dataset would quantify the contribution of each change and would test whether the ranking produced by the third stage predicts the observed ordering of effects. A broader case collection spanning several stacks would test whether the 2 diagnostic regimes exhaust the space, and formalizing the prioritization rule into a computable score over profiler output would move the third stage toward automation.

5. Conclusion

Latency degradation in long-lived server systems arises from the accumulation of data against design decisions taken when the data was small, and it arrives without an event that monitoring records as a cause. This article organizes server-side profiling into a proactive procedure of 4 stages that establishes a baseline, localizes the delay using an instrument selected based on the available evidence, ranks findings by share of total time and invocation frequency, and applies a scoped change verified by re-measurement.

The procedure was exercised in both of its diagnostic regimes. Where a single statement dominated a request, plan analysis exposed a recursive traversal embedded in the main query, gaps in index coverage, a hash operation spilling to disk after a memory limit was crossed, and expensive columns computed for discarded rows, and the 4 corresponding changes reduced endpoint response time from roughly 10 seconds to roughly 500 milliseconds. Where no single operation dominated, call-level profiling attributed the delay to derived-data recalculations repeated once per saved item, and deferring them into a single pass removed the repetition.

Both defects share a property that motivates the proactive framing. Additional hardware would have concealed each of them while leaving the mechanism intact, and that concealment would expire at the next threshold of data growth. The economic implication follows from the pricing of the two responses: capacity is billed per unit of time and grows with the workload that an unremoved defect generates, whereas a diagnosis-led change is paid for once in engineering hours. The value of the procedure lies in its repetition, since a baseline captured while an endpoint still performs acceptably converts a future slowdown into a comparison against a known quantity.

The contribution is a modest one. The instruments used are established, and the optimizations applied are documented practice, and what the cases demonstrate is that a small ordered sequence connects a measurement to a verified outcome across defect classes that behave differently under observation. The scope is 2 cases on one platform, and the transferability argued here awaits testing on the wider population of stacks and workloads to which the stages appear to apply.

Acknowledgements

None.

Funding Information

This research received no external funding.

Author Contributions

Ilia Detinkin: conceptualization, methodology, investigation, analysis of the cases, writing of the original draft, review and editing. The author has read and approved the final version of the manuscript.

Conflict of Interest

The author declares no conflict of interest. The cases were anonymized, and no detail permits attribution to

a specific employer or customer.

REFERENCES

1. A. Janes, X. Li, and V. Lenarduzzi, "Open tracing tools: Overview and critical comparison," Journal of Systems and Software, vol. 204, Art. no. 111793, 2023, DOI: 10.1016/j.jss.2023.111793.

2. B. Li, X. Peng, Q. Xiang, H. Wang, T. Xie, J. Sun, and X. Liu, "Enjoy your observability: An industrial survey of microservice tracing and analysis," Empirical Software Engineering, vol. 27, no. 1, Art. no. 25, 2022, DOI: 10.1007/s10664-021-10063-9.

3. W. Zhou, C. Lin, X. Zhou, and G. Li, "Breaking it down: An in-depth study of index advisors," Proceedings of the VLDB Endowment, vol. 17, no. 10, pp. 2405–2418, 2024, DOI: 10.14778/3675034.3675035.

4. J. Shi, G. Cong, and X.-L. Li, "Learned index benefits: Machine learning based index performance estimation," Proceedings of the VLDB Endowment, vol. 15, no. 13, pp. 3950–3962, 2022, DOI: 10.14778/3565838.3565848.

5. B. A. Muse, K. W. Nafi, F. Khomh, and G. Antoniol, "Data-access performance anti-patterns in data-intensive systems," Empirical Software Engineering, vol. 29, Art. no. 144, 2024, DOI: 10.1007/s10664-024-10535-8.

6. W. Liu, S. Mondal, and T.-H. Chen, "An empirical study on the characteristics of database access bugs in Java applications," ACM Transactions on Software Engineering and Methodology, vol. 33, no. 7, Art. no. 181, 2024, DOI: 10.1145/3672449.

7. C. Laaber, H. C. Gall, and P. Leitner, "Applying test case prioritization to software microbenchmarks," Empirical Software Engineering, vol. 26, no. 6, Art. no. 133, 2021, DOI: 10.1007/s10664-021-10037-x.

8. M. Grambow, D. Kovalev, C. Laaber, P. Leitner, and D. Bermbach, "Using microbenchmark suites to detect application performance changes," IEEE Transactions on Cloud Computing, vol. 11, no. 3, pp. 2575–2590, 2023, DOI: 10.1109/TCC.2022.3217947.

9. R. Mehra, P. Pathania, V. S. Sharma, V. Kaulgud, S. Podder, and A. P. Burden, "Assessing the impact of refactoring energy-inefficient code patterns on software sustainability: An industry case study," in Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE 2023), pp. 1825–1827, 2023, DOI: 10.1109/ASE56229.2023.00205.

10. C. Zhang, Z. Dong, X. Peng, B. Zhang, and M. Chen, "Trace-based multi-dimensional root cause localization of performance issues in microservice systems," in Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE 2024), Art. no. 110, pp. 1–12, 2024, DOI: 10.1145/3597503.3639088.