Algorithm Placement in Full-Stack Applications Across Data, Business Logic, and Presentation Layers

Max Angelo D. Perin
Bohol Island State University

Author Note

[Correspondence concerning this article should be addressed to Max Angelo D. Perin, Bohol Island State University. Email: `maxangel.perin@bisu.edu.ph`.]

Abstract

Full-stack applications are commonly structured into data access, business logic, and presentation layers, with software architecture guidance emphasizing clear separation of concerns among these tiers. At the same time, algorithm analysis and Big-O notation are typically introduced under a uniform-cost model that does not distinguish between database operations, in-memory computation, and user interface rendering. This study uses a mathematical approach to connect these two perspectives. A generic nested-loop pattern representative of quadratic-time behavior is analyzed and its exact step count is derived as $S(n) = (n^2 - n)/2$. This step count is then reinterpreted in three cost models corresponding to the data access layer (indexed SQL `SELECT` and `UPDATE` operations in a stored procedure), the business logic layer (in-memory array operations in an application server), and the presentation layer (JavaScript computation together with virtual DOM diffing and DOM updates in a React-style framework). Explicit running-time expressions are obtained for each layer, showing that the same nested-loop algorithm has $\Theta(n^2)$ behavior in the business and presentation layers and can reach $\Theta(n^2 \log n)$ in the data access layer when index traversal costs are considered. Because composite database steps are much more expensive than in-memory operations and presentation layer rendering adds additional overhead, the analysis supports complexity-aware guidelines for algorithm placement: declarative, set-based queries and integrity constraints should remain in the data access layer; algorithmic control flow over large datasets should be concentrated in the business logic layer; and the presentation layer should be reserved for comparatively lightweight computations that preserve interactive performance.

Keywords: full-stack architecture; algorithm analysis; data access layer; business logic layer; presentation layer

1 Introduction

Layered and three-tier architectures are widely used in enterprise and web development to separate concerns and improve maintainability. IBM (n.d.) describes a three-tier architecture as a structure with a presentation tier that manages user interaction, an application tier where processing and business rules reside, and a data tier responsible for storage and management. Techtarget (2024) similarly defines the three-tier application architecture as a modular client-server design consisting of presentation, application, and data tiers, with the aim of improving scalability and manageability. In the context of enterprise systems, Fowler (2003) describes patterns for dividing applications into presentation, domain logic, and data source layers, emphasizing that placing inappropriate responsibilities in a layer, such as heavy domain logic in the data source, leads to brittle designs. Microsoft's modern web application guidance positions the application layer as the primary location to implement use cases and coordinate data access, while the data layer focuses on persistence, and the user interface layer focuses on rendering and interaction (Microsoft, 2023).

Although these references outline clear responsibilities for each layer, practitioners frequently encounter uncertainty about where concrete algorithmic logic should reside. A recurring example is the debate over business rules in stored procedures compared with business rules in application code. Some authors highlight that stored procedures can centralize logic close to the data and reduce network round-trips, while others argue that complex logic in the database increases coupling, complicates testing, and constrains evolution (Mulki, 2025; OWOX, 2025). Similar debates arise around client-side versus server-side processing. Guidance often mentions performance, security, and user experience, but typically relies on qualitative arguments rather than a formal cost model that compares the same algorithm implemented in different layers (Iskandar, Wahyudi, & Puspitasari, 2024; Yegorova, 2023).

In parallel, algorithm analysis and Big-O notation are central in computer science theory and education. Cormen, Leiserson, Rivest, and Stein (2022) present an asymptotic analysis under a random-access machine model, where each primitive operation has a uniform cost, and show that nested-loop patterns typically lead to quadratic-time complexity. Introductory tutorials from GeeksforGeeks (2025) and Programiz (n.d.) explain the Big-O, Big-Theta, and Big-Omega notations and classify algorithms according to their growth rates. These treatments habitually abstract away constant factors and environment-specific costs. This abstraction is valuable for comparing algorithms in the same environment, but it does not distinguish between the cost of a "step" in a database engine, an application server, or a browser runtime. As a result, two implementations of the same nested-loop algorithm — one expressed as repeated indexed SQL statements and one expressed as in-memory array operations — may both be labeled as quadratic, even though their practical behavior differs significantly.

Modern web frameworks make these differences particularly visible. Microsoft's architecture guidance for ASP.NET Core and Azure presents common web application architectures where the application layer is responsible for domain logic and orchestration, the data layer focuses on persistence and query execution, and the presentation layer manages user interaction and display (Microsoft, 2023). React's documentation explains that the framework maintains a virtual DOM, an in-memory representation of the user interface, and reconciles it with the actual DOM through a diffing and update algorithm (React Team, n.d.). Yadav (2025) notes that this process is optimized, but still incurs a cost that increases with the number of elements updated during each rendering. Studies comparing client-side and server-side rendering observe that rendering large lists on the client can negatively affect performance if heavy computations are coupled to each render (Iskandar et al., 2024).

These considerations motivate a mathematical investigation of algorithm placement in full-stack applications. Instead of treating algorithm analysis and layered architecture as separate subjects, it is useful to construct explicit cost models for the data access, business logic, and presentation layers and to examine how a representative nested-loop algorithm behaves under each model. By deriving exact step counts and combining them with layer-specific costs, it becomes possible to obtain closed-form running-time expressions and to reason formally about where algorithmic logic should reside. The present study follows this approach for a canonical quadratic pattern and uses the resulting expressions to propose complexity-aware guidelines for algorithm placement across the data access, business logic, and presentation layers.

2 Methodology

The study employs a theoretical methodology based on explicit cost modeling and step counting. The central idea is to analyze a representative nested-loop algorithm in three different execution environments that correspond to the data access, business logic, and presentation layers, and then to compare the resulting running-time expressions.

The starting point is a generic nested-loop pattern that captures quadratic-time behavior in a compact, language-independent form. This pattern can be written in pseudocode as follows:

```
NESTED-LOOP(ALIST[1..n]):
    for i ← 1 to n - 1 do
        for j ← 1 to n - i do
            PROCESS(ALIST, i, j)
```

The subroutine `PROCESS` represents a constant amount of work on elements associated with indices i and j, such as comparing two values and optionally swapping them, or evaluating a condition on the pair. For each fixed i, the inner loop executes $n-i$ times. The total number of times `PROCESS` is called is therefore

$$S(n) = \sum_{i=1}^{n-1}(n-i).$$

Reindexing this sum by setting $k = n - i$ yields

$$S(n) = \sum_{k=1}^{n-1} k,$$

which is the standard arithmetic series whose closed form is

$$S(n) = \frac{(n-1)n}{2} = \frac{n^2 - n}{2}.$$

This expression provides an exact step count for any algorithm that conforms to the nested-loop structure. The asymptotic order is quadratic, $S(n) \in \Theta(n^2)$, but the exact form is preserved because it will be combined with different per-step costs in each layer.

To move from this abstract pattern to concrete full-stack environments, the methodology instantiates the nested-loop structure in three ways: as a SQL stored procedure, as a Node.js function, and as a React-based front-end handler.

In the data access layer, the nested loops are implemented inside the database as a stored procedure that performs comparisons and swaps using SQL operations on a table:

```
CREATE TABLE numbers (
    idx   INT PRIMARY KEY,
    value INT
);


DELIMITER $$


CREATE PROCEDURE bubble_sort_numbers()
BEGIN
    DECLARE i INT DEFAULT 1;
    DECLARE j INT;
    DECLARE n INT;
    DECLARE vj INT;
    DECLARE vj1 INT;

    SELECT COUNT(*) INTO n FROM numbers;

    WHILE i <= n - 1 DO
        SET j = 1;
        WHILE j <= n - i DO
            SELECT value INTO vj
            FROM numbers
            WHERE idx = j;

            SELECT value INTO vj1
            FROM numbers
            WHERE idx = j + 1;

            IF vj > vj1 THEN
                UPDATE numbers SET value = vj1 WHERE idx = j;
                UPDATE numbers SET value = vj  WHERE idx = j + 1;
            END IF;

            SET j = j + 1;
        END WHILE;
        SET i = i + 1;
    END WHILE;
END$$
```

```
DELIMITER ;
```

In this implementation, each inner iteration issues two indexed `SELECT` statements and, in the worst case, two indexed `UPDATE` statements. A composite primitive step in the data access layer is defined to consist of these operations. Under a simple model, this composite step is assigned a constant cost $c_{\text{DA,step}}$. Under a refined model, the cost of each indexed operation is taken to be proportional to $\log n$, so that the composite step has cost on the order of $c_{\text{DA,step}} \log n$.

In the business logic layer, the nested loops are implemented as a Node.js function operating on an in-memory array:

```
function bubbleSort(arr) {
  const a = [...arr];         // copy to avoid mutating caller's array
  const n = a.length;

  for (let i = 0; i < n - 1; i++) {
    for (let j = 0; j < n - i - 1; j++) {
      if (a[j] > a[j + 1]) {
        const temp = a[j];
        a[j] = a[j + 1];
        a[j + 1] = temp;
      }
    }
  }
  return a;
}
```

Here, each inner iteration performs two array reads, one comparison, and in the worst case three assignments. A primitive step in the business logic layer is defined to consist of this fixed sequence of in-memory operations, with constant cost $c_{\mathrm{BL,step}}$ under the random-access memory model.

In the presentation layer, the same nested loops are invoked from a React component that manages UI state:

```
function bubbleSortByValue(list) {
  const a = [...list];
  const n = a.length;

  for (let i = 0; i < n - 1; i++) {
    for (let j = 0; j < n - i - 1; j++) {
      if (a[j].value > a[j + 1].value) {
        const temp = a[j];
        a[j] = a[j + 1];
        a[j + 1] = temp;
      }
    }
  }
  return a;
}

function NumberList({ initialData }) {
  const [data, setData] = React.useState(initialData);

  const handleSort = () => {
    const sorted = bubbleSortByValue(data);
    setData(sorted);
  };
  return (
    <>
      <button onClick={handleSort}>Sort</button>
      <ul>
        {data.map(item => (
          <li key={item.id}>{item.value}</li>
        ))}
      </ul>
    </>
  );
}
```

On each click of the sort button, the component performs the nested-loop computation in JavaScript and then triggers a re-render of the list. The primitive computational step has constant cost $c_{\mathrm{PL,step}}$. In addition, rendering a list of n elements is modeled as requiring $d_{\mathrm{PL}}n$ work due to virtual DOM diffing and DOM updates, consistent with descriptions of React's rendering process (React Team, n.d.; Yadav, 2025).

With these implementations in place, the methodology combines the common step count $S(n)$ with the layer-specific step costs and additional overhead to obtain explicit running-time functions. In the data access layer under the simple constant-cost model,

$$T_{\mathrm{DA}}(n) = c_{\mathrm{DA,step}} \cdot S(n) + O(n) = c_{\mathrm{DA,step}} \cdot \frac{n^2 - n}{2} + O(n),$$

which can be written as

$$T_{\mathrm{DA}}(n) = A_{\mathrm{DA}}n^2 + B_{\mathrm{DA}}n + C_{\mathrm{DA}},$$

with $T_{\mathrm{DA}}(n) \in \Theta(n^2)$. Under the refined index-cost model, the composite step has cost proportional to $\log n$, which yields

$$T_{\mathrm{DA}}(n) = c_{\mathrm{DA,step}} \log n \cdot \frac{n^2 - n}{2} + O(n) \in \Theta(n^2 \log n).$$

In the business logic layer, the running time is

$$T_{\mathrm{BL}}(n) = c_{\mathrm{BL,step}} \cdot S(n) + O(n) = c_{\mathrm{BL,step}} \cdot \frac{n^2 - n}{2} + O(n) = A_{\mathrm{BL}} n^2 + B_{\mathrm{BL}} n + C_{\mathrm{BL}},$$

which belongs to $\Theta(n^2)$.

In the presentation layer, the running time combines computation and rendering:

$$T_{\mathrm{PL}}(n) = c_{\mathrm{PL,step}} \cdot S(n) + d_{\mathrm{PL}} n + O(1) = c_{\mathrm{PL,step}} \cdot \frac{n^2 - n}{2} + d_{\mathrm{PL}} n + O(1) = A_{\mathrm{PL}} n^2 + D_{\mathrm{PL}} n + C_{\mathrm{PL}},$$

which is also in $\Theta(n^2)$ but includes a linear term tied directly to rendering.

The final part of the methodology interprets these expressions under the assumption that composite database steps are significantly more expensive than in-memory steps, and that rendering in the presentation layer has nontrivial cost. This interpretation leads to an ordering of the running times and forms the basis for the discussion of algorithm placement.

3 Results and Discussion

The cost functions derived from the nested-loop pattern exhibit both shared structure and important differences across the three layers. In all cases, the computation performs

$$S(n) = \frac{n^2 - n}{2}$$

primitive steps, reflecting the quadratic nature of the nested loops. The differences arise in the cost assigned to each step and in any additional overhead associated with the environment.

In the business logic layer, the running time takes the form $T_{\text{BL}}(n) = A_{\text{BL}}n^2 + B_{\text{BL}}n + C_{\text{BL}}$, with A_{BL} proportional to the cost of a fixed sequence of in-memory operations. Under standard random-access assumptions, this leads to $\Theta(n^2)$ behavior with relatively small constants (Cormen et al., 2022). In the presentation layer, the running time $T_{\text{PL}}(n) = A_{\text{PL}}n^2 + D_{\text{PL}}n + C_{\text{PL}}$ includes both the quadratic computational term and a linear rendering term, which becomes visible in user-perceived responsiveness as n increases (React Team, n.d.; Yadav, 2025). In the data access layer, the running time $T_{\text{DA}}(n) = A_{\text{DA}}n^2 + B_{\text{DA}}n + C_{\text{DA}}$ is at least quadratic and may behave as $\Theta(n^2 \log n)$ when index traversal is modeled, reflecting the overhead of query parsing, planning, logging, and index operations.

These relationships can be summarized in a comparative table. The table uses a simple constant-cost index model for the data access layer but notes the effect of a refined $\log n$ cost.

Assuming that $c_{\text{DA,step}}$ is substantially larger than $c_{\text{BL,step}}$ and that d_{PL} is non-negligible, the running times satisfy

$$T_{\text{DA}}(n) > T_{\text{PL}}(n) > T_{\text{BL}}(n)$$

for sufficiently large n. This ordering means that nested-loop logic over large datasets is most expensive when executed in the data access layer, least expensive in the business logic layer, and intermediate in the presentation layer. Architectural texts advise against placing extensive domain logic in the database and encourage the use of application layers for business rules (Fowler, 2003; Microsoft, 2023). The computations here provide a formal explanation for that advice by showing how the underlying cost models differ.

The results also have implications for client-side design. Studies on client-side versus server-side rendering note that heavy computations and frequent updates on large client-side collections can degrade performance (Iskandar et al., 2024). The presence of the rendering term $d_{\text{PL}}n$ in the cost function for the presentation layer makes this effect explicit. Even when the computational cost matches that of the business logic layer, the additional linear term tied to rendering can push the overall cost beyond acceptable limits for interactive use.

This supports recommendations to move heavy computations to the server and to keep the client focused on interaction, presentation, and incremental updates.

The analysis also clarifies the role of duplication across layers. Some duplication of simple checks across client and server is unavoidable and desirable for usability and security; such checks are typically linear or constant-time and operate on relatively small inputs. When logic exhibits quadratic behavior, however, the derived cost functions suggest that it should be centralized in the business logic layer rather than being replicated in stored procedures and client-side handlers. This avoids paying the same high-order cost multiple times in different environments.

Beyond its architectural implications, the analysis has educational value. By presenting a familiar nested-loop pattern in three different cost models, the study provides a concrete bridge between abstract asymptotic analysis and practical full-stack design decisions. Students and practitioners can see how a single algorithm with a fixed step count leads to different running-time expressions depending on whether the primitive operations are database lookups, in-memory computations, or UI updates, reinforcing the idea that complexity analysis should inform not only which algorithms are chosen but also where they are placed in an application.

4 Conclusion

The study investigated algorithm placement in full-stack applications by analyzing a representative nested-loop pattern across the data access, business logic, and presentation layers. A generic nested-loop pseudocode structure was considered, and its exact step count was derived as $S(n) = (n^2 - n)/2$. This count was then reinterpreted in three cost models: one where the primitive step is a composite database operation consisting of indexed `SELECT` and `UPDATE` statements; one where the primitive step is a small fixed set of in-memory operations on arrays; and one where the primitive step is the same in-memory computation but followed by a rendering phase that updates a list of user interface elements.

The resulting running-time expressions showed that the same nested-loop algorithm exhibits $\Theta(n^2)$ behavior in the business logic and presentation layers and at least $\Theta(n^2)$, and potentially $\Theta(n^2 \log n)$, in the data access layer when index traversal costs are modeled. Under realistic assumptions about the relative sizes of the constants, the data access implementation is most expensive, the business logic implementation is least expensive, and the presentation implementation lies between the two due to the added rendering term. These findings provide mathematical support for longstanding architectural advice that databases should focus on declarative, set-based queries and integrity constraints, application layers should implement domain logic and workflows, and user interfaces should concentrate on interaction and presentation rather than heavy computation (Fowler, 2003; IBM, n.d.; Microsoft, 2023).

The analysis is theoretical and deliberately simplifies several aspects of real systems. Network latency, concurrency, caching, and hardware variability are not modeled. The study focuses on a single family of algorithms characterized by nested loops and a single input-size parameter, and it does not include empirical timing measurements on actual technology stacks. Future work can extend the cost models to other algorithmic structures, incorporate multiple input parameters, and validate the theoretical predictions with experiments. Additional research may also explore how these complexity-aware placement guidelines can be integrated into teaching materials for courses on algorithms, databases, and web development, and how automated tools could leverage similar models to flag potentially problematic algorithm placement during code review.

5 Recommendations

The theoretical results suggest several practical and educational recommendations. In practice, algorithmic logic that follows a nested-loop or other quadratic pattern over large datasets should be implemented in the business logic layer, where in-memory operations are relatively cheap and algorithmic improvements can be applied more easily. Databases should be used primarily for declarative, set-based operations, such as filtering, joining, grouping, and ordering, and should avoid explicit row-by-row loops for large tables. Stored procedures, when used, should remain focused on atomic data operations and simple validations rather than implementing complex algorithms.

For the presentation layer, applications should avoid repeatedly performing quadratic computations over large collections of user interface elements. When large lists must be sorted, matched, or validated, the heavy computation should be carried out on the server and the client should receive data that is already in a form suitable for efficient rendering. Client-side logic should emphasize interaction, formatting, and light filtering or transformation on data that has been preprocessed in the business logic layer.

In education, the same nested-loop example can be presented in the context of different layers to show that a single Big-O classification can correspond to different practical behaviors depending on the execution environment. Courses on algorithms can refer to these cost models when explaining that constant factors and underlying primitives matter, while courses on software architecture and web development can use the models to justify layering decisions and to teach students to consider both asymptotic complexity and placement when designing systems. Assignments that ask learners to derive and compare running-time expressions for the same logic across different layers can help develop a more integrated understanding of complexity and architecture.

References

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to algorithms* (4th ed.). MIT Press.

Fowler, M. (2003). *Patterns of enterprise application architecture.* Addison-Wesley.

GeeksforGeeks. (2025, August 27). Big O notation tutorial – A guide to Big O analysis. Retrieved from https://www.geeksforgeeks.org/dsa/analysis-algorithms-big-o-analysis/

IBM. (n.d.). What is three-tier architecture? Retrieved from https://www.ibm.com/think/topics/three-tier-architecture

Iskandar, A., Wahyudi, A., & Puspitasari, N. (2024). A comparative review of server rendering and client-side rendering in web applications. *International Journal of Scientific Research in Engineering and Technology, 10*(2), 218–225.

Microsoft. (2023). Architect modern web applications with ASP.NET Core and Azure. Retrieved from https://learn.microsoft.com/dotnet/architecture/modern-web-apps-azure

Mulki, R. (2025, April 25). Stored procedures vs. application logic: When to use which? Retrieved from https://rizqimulki.com/stored-procedures-vs-application-logic-when-t

OWOX. (2025, April 14). Stored procedures in SQL: Benefits, examples and use cases. Retrieved from https://www.owox.com/blog/articles/stored-procedures-sql-benefits-examples-

Programiz. (n.d.). Asymptotic analysis: Big-O notation and more. Retrieved from https://www.programiz.com/dsa/asymptotic-notations

React Team. (n.d.). Virtual DOM and internals. Retrieved from https://legacy.reactjs.org/docs/faq-internals.html

Techtarget. (2024, October 22). What is a 3-tier application architecture? Retrieved from https://www.techtarget.com/searchsoftwarequality/definition/3-tier-application

Yadav, P. (2025, August 13). ReactJS virtual DOM. Retrieved from https://www.geeksforgeeks.org/reactjs/reactjs-virtual-dom/

Yegorova, N. (2023). Layers in software architecture and separation of concerns in web applications. Retrieved from https://medium.com/

Table 1: Comparison of nested-loop cost across layers.

Aspect	Data Access Layer (SQL)	Business Logic Layer (Node.js)	Presentation Layer (React)
Data representation	Rows in table `numbers(idx, value)`	In-memory array `a[0..n-1]`	React state array rendered as HTML list
Example implementation	Stored procedure `bubble_sort_numbers()` with `WHILE` and `SELECT`/`UPDATE`	Function `bubbleSort(arr)` with nested `for` loops	`bubbleSortByValue` plus `NumberList` component calling `setState` and re-rendering
Primitive step	Bounded number of indexed `SELECT` and `UPDATE` operations	Bounded number of array reads, comparisons, and assignments	Same computation as BL plus later virtual DOM diffing and DOM updates
Step cost (simple model)	$c_{\mathrm{DA,step}}$	$c_{\mathrm{BL,step}}$	$c_{\mathrm{PL,step}}$
Additional overhead	Query parsing, planning, logging, transactions	Loop control and function-call overhead	Rendering cost $R_{\mathrm{PL}}(n) = d_{\mathrm{PL}}n$
Running-time function	$T_{\mathrm{DA}}(n) = c_{\mathrm{DA,step}}\frac{n^2-n}{2} + O(n)$	$T_{\mathrm{BL}}(n) = c_{\mathrm{BL,step}}\frac{n^2-n}{2} + O(n)$	$T_{\mathrm{PL}}(n) = c_{\mathrm{PL,step}}\frac{n^2-n}{2} + d_{\mathrm{PL}}n + O(1)$
Asymptotic (simple)	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Asymptotic (with index)	$\Theta(n^2 \log n)$	$\Theta(n^2)$	$\Theta(n^2)$
Relative constants	Largest (DB engine overhead)	Smallest (in-memory ops)	Medium (in-memory + rendering)
Practical effect for large n	Very slow, resource-intensive	Slow but acceptable for server-side work	Slow and visibly laggy for large lists
Recommended role	Declarative set-based queries and integrity constraints	Algorithmic control flow on large datasets	Lightweight client-side transformations and rendering