CHAPTER 1

INTRODUCTION

This chapter delves into the essential aspects of the study, beginning with the underlying motivations that drive the research. It progresses through an examination of the theoretical and conceptual frameworks that anchor the study, articulates the specific problems being addressed, and, where applicable, outlines the hypotheses and assumptions involved. It also details the scope and limitations of the research, concluding with an analysis of the significance of the findings. Each component is crafted to enhance understanding and provide a cohesive structure to the exploration of the topic.

1.1 Background of the Study

In recent years, Python-based libraries and frameworks have played a pivotal role in advancing software development, particularly in data manipulation and management[1]. However, as an interpreted language, Python generally suffers from slower execution times compared to compiled languages[10]. This drawback becomes critical in performance-sensitive components like Object-Relational Mapping (ORM) frameworks. Interpreted execution models process code line-by-line at runtime, which introduces overhead for operations that are repeated frequently—such as converting large volumes of database rows into Python objects during query execution. This overhead can lead to significant latency and memory inefficiencies, especially in high-traffic applications or batch processing systems.

For example, in ORM scenarios that involve retrieving thousands of records (e.g., fetching user posts or comments in a social media application), each row must be parsed and mapped into objects repeatedly using runtime interpretation. This not only increases CPU load but also results in inconsistent memory usage due to the lack of pre-optimization. Furthermore, interpreted execution limits opportunities for low-level optimization techniques like instruction pipelining or inlining, which are common in compiled environments. Therefore, applying Just-In-Time (JIT) compilation to the ORM data mapping process becomes a logical enhancement—it allows frequently used object-mapping routines to be compiled into native machine instructions, significantly reducing interpretation overhead and boosting execution efficiency.

Similar JIT-based approaches have been implemented in other programming environments, such as ThriveJIT in Java for expression evaluation[30], JITDB using WebAssembly for real-time query execution in JavaScript[33], and PyPy or Numba in Python for accelerating numerical computations through JIT compilation[22]. These implementations highlight the cross-language relevance and effectiveness of embedding compilation layers within high-level frameworks to improve runtime performance.

Python ORM frameworks simplified database interactions by abstracting SQL queries and managing schemas, thereby enhancing developer productivity and maintainability[5]. Frameworks such as SQLAlchemy, Pony ORM, and Tortoise ORM were widely adopted in Python-based systems due to their intuitive APIs and high-level abstraction capabilities. However, this convenience often comes with trade-offs, particularly in the context of large datasets or complex relational queries[19]. The abstraction layers introduce additional runtime processing, especially when converting query results into nested Python object hierarchies. As applications grow in scale—such as modern social media platforms like Instagram, which reported over two billion monthly active users by 2024[4]—the volume of data and the complexity of data access patterns increase dramatically.

This highlights the motivation behind this research: interpreted ORM frameworks, while developer-friendly, often struggle with performance bottlenecks in data-intensive environments. The need for faster object-relational data mapping becomes critical in scenarios involving millions of CRUD operations per day, like in Instagram's comment, like, or follow systems. By proposing JITORM, this research addresses such performance concerns by integrating Just-In-Time compilation into the ORM layer—offering a promising alternative for scalable and efficient data management in large-scale applications.

Despite Python's advantages in rapid development, existing ORM frameworks still depend heavily on interpreted execution paths, which become a bottleneck in data-intensive scenarios [21]. This issue is exacerbated by the absence of low-level optimization techniques like Just-In-Time (JIT) compilation. While some interpreter-level improvements exist—such as PyPy's JIT-enabled interpreter—they typically target general-purpose execution rather than optimizing specific workloads like ORM data mapping. Consequently, operations such as batch inserts, object hydration during bulk queries, or complex joins can suffer from increased latency and memory overhead.

JIT compilation addresses these issues by transforming high-level instructions into efficient machine code at runtime, reducing the need for repeated interpretation. In the context of ORM, integrating JIT specifically into the data mapping phase offers a tangible performance advantage by eliminating redundant conversion logic and improving CPU and memory utilization. Studies such as those by Lam et al. on Numba [22], and developments in TorchDynamo for PyTorch[23], demonstrate how domain-specific JIT techniques can yield significant performance gains when targeted to computational bottlenecks.

To operationalize this idea, a custom ORM framework called **JITORM** is developed in this study. JITORM introduces a modular architecture comprising three core components: `Model` definitions to describe database schemas, `Session` objects to manage database connections and transactions, and a `JITCompiler` module that converts schema-aware mapping logic into LLVM Intermediate Representation (IR) using `llvmlite`, and compiles it into optimized native machine code. The use of `llvmlite` is preferred over alternatives like PyPy or Numba because it offers fine-grained control over the compilation pipeline

and is well-suited for integrating domain-specific compilation logic, especially at the data-mapping layer.

This research evaluates the performance of JITORM in comparison with widely used Python ORM libraries—namely SQLAlchemy, Pony ORM, and Tortoise ORM—to measure its practical benefits across varying data volumes and operation types. The comparative analysis focuses on execution time, memory usage, and CPU utilization under controlled, resource-constrained environments to simulate realistic deployment scenarios.

The significance of this study lies in demonstrating that domain-specific JIT compilation can be effectively integrated into high-level frameworks like ORM, potentially opening new design patterns for Python performance engineering. By providing empirical evidence on where and how JIT can outperform traditional interpreted models, this research contributes both practically and academically to the field of software performance optimization in interpreted languages.

While JIT compilation offers performance advantages in high-load scenarios, it may introduce overhead in lightweight or one-off operations. Therefore, identifying the contexts in which JIT integration is beneficial—or counterproductive—is essential to ensure its practical effectiveness.

1.2 Research Question

1. To what extent does the integration of Just-In-Time (JIT) compilation into Python ORM frameworks improve performance compared to standard ORM frameworks across different scenarios (data scale and operation types)?
2. In what scenarios does the use of JIT in ORM frameworks become inefficient or counterproductive, particularly in terms of execution time, memory usage, and CPU consumption?
3. How can JITORM be integrated into modern Python web frameworks such as Flask or FastAPI?

1.3 Objectives

This research aims to evaluate the impact of integrating Just-In-Time (JIT) compilation into Python-based Object-Relational Mapping (ORM) frameworks, particularly in data-intensive environments and under constrained computing resources. The study focused on operations involving large datasets and repeated data processing tasks, such as bulk insertion and data list retrieval, to determine whether the use of JIT compilation can improve execution time, CPU usage, and memory efficiency. By benchmarking the proposed JIT-based ORM framework (JITORM) against widely used native ORM frameworks—SQLAlchemy, Pony ORM, and Tortoise ORM—this research sought to demon-

strate the performance advantages and limitations of JIT integration in real-world scenarios.

1.4 Hypotheses

A JIT-integrated ORM framework will demonstrate higher efficiency in data processing tasks—by reducing execution time and optimizing CPU and memory resource usage—compared to native Python ORM frameworks. It is hypothesized that this performance advantage will be particularly evident in operations involving large datasets or repetitive access patterns, while its benefits may be less prominent or even counterproductive in small-scale or single-record operations due to compilation overhead.

1.5 Scope And Limitations

This study is limited to the design, implementation, and evaluation of a custom ORM framework (JITORM) that integrates Just-In-Time (JIT) compilation at the data mapping stage. The framework is evaluated based on core CRUD operations—create (insert), read (get single, get list), update, and delete—executed on synthetically generated datasets ranging in size from 50,000 to 200,000 records. These operations were tested across two resource-constrained environments (0.2 CPU & 512MB memory and 0.5 CPU & 1024MB memory) to simulate typical low-resource deployment conditions.

The analysis distinguishes between high-volume operations (e.g., inserting or retrieving many records) and low-volume operations (e.g., retrieving or updating a single record), to assess where JIT offered meaningful performance gains and where it may be inefficient. Advanced ORM features such as complex joins, relationships, or transactional consistency are excluded to maintain a focused and controlled evaluation on core performance aspects.