Instituto Superior
de Engenharia
Politécnico de Coimbra

DEPARTMENT OF SYSTEMS AND COMPUTER ENGINEERING

Evaluation of the Impact of AES Encryption on Query Read Performance Across Oracle, MySQL, and SQL Server Databases

Dissertation to fulfil the Master's degree in Informatics Engineering

Specialization in Software Engineering

Author

Márcio André Correia de Carvalho

Supervisors

Prof. Doutor Jorge Fernandes Rodrigues Bernardino

Prof. Doutor Filipe Alexandre Almeida Ningre de Sá

Departamento de Engenharia Informática e de Sistemas

Instituto Superior de Engenharia de Coimbra

INSTITUTO POLITÉCNICO
DE COIMBRA

INSTITUTO SUPERIOR
DE ENGENHARIA
DE COIMBRA

Coimbra, Julho 2025

ACKNOWLEDGEMENTS

I would like to express my sincere appreciation and gratitude to my supervisor, Prof. Doutor Jorge Bernardino, for his persistence, guidance, and feedback. His experience and critical questions pushed me to sharpen my thinking and was extraordinarily important in the completion of this thesis and for my academic and personal growth.

I am also extremely thankful to my co-supervisor, Prof. Doutor Filipe Sá, who played a key role in the development of this work, giving insightful feedback, encouragement, and inspiration. His extensive knowledge, approachability, and the lightness with which he guided me were a true source of inspiration and crucial in the completion of this work.

In addition, I wish to acknowledge the Instituto Superior de Engenharia de Coimbra and the IT support team for providing me with the necessary academic tools.

Lastly, I must express my gratitude to everyone who helped me during this entire journey. I am thankful to my family, especially to my parents Maria Correia and Paulo Carvalho and to my brother Dinis Carvalho for their support and for making every effort from an early age to give me the best education and tools to pursue all my dreams and goals. I also want to say thanks to my closest friends, who always gave me motivation and encouragement throughout this journey. I am also thankful to the company, Paua, which supported me during this process, in particular to one of its founders and my friend, André Pinho, whose guidance and motivation were essential at every stage of this journey as well. Finally, to my girlfriend, Inês Rocha, thank you for your patience, love, and unshakable faith in me.

ABSTRACT

Data security is essential when information is shared between entities, as vulnerabilities can compromise both the sender and the receiver, and even entire systems. This thesis evaluates how encryption affects the database performance during query execution.

To achieve that, we started by identifying encryption algorithms that were natively supported and consistently implemented across multiple database platforms. Based on these criteria, AES was selected as the most suitable algorithm due to its wide adoption and support across platforms, in AES-128, AES-192, and AES-256 bit variants. Then it was selected which databases supported these algorithms and set up an environment for each scenario, which included a different encryption algorithm, scale factor and database platforms.

For each scenario, standard benchmark (TPC-H) queries were run to see how the system performed. The elapsed times obtained from the execution tests were processed through a Python script, which converted the values from a raw text file to a properly formatted Excel spreadsheet prepared for analysis, displaying the time from the respective query number to the respective execution number.

The evaluation parameters selected were the execution time and the system resources' CPU, RAM, and disk activity. Execution time was chosen as the primary performance metric, as it directly reflects the impact of encryption on process efficiency. System resources such as CPU, RAM and disk activity were selected to measure if the system conditions were similar for the tests. A high variation in the performance results was observed across different configurations, especially in Oracle, including in some non-encryption baselines.

The study analyzed Oracle, MySQL, and SQL Server performance when querying encrypted databases across multiple scale factors. Oracle showed that AES-192 is the most balanced algorithm at smaller scale factors, but AES-128 outperformed the other AES key sizes at a scale factor of 10. SQL Server showed an expected performance scaling, with AES-192 being the most balanced again, as in Oracle, but at scale factor 1, AES-256 performed better. MySQL v8.0.38 showed reduced variation at larger scale factors, though overall stability improved significantly in enterprise version 9.2. When comparing SQL Server to Oracle, SQL Server performed much better than Oracle, at a scale factor of 10, AES-256 outperformed Oracle up to 40 times faster, even without encryption, Oracle was about 8 times slower than SQL Server. MySQL v8.0.38 had more result variations in the two tested scale factors, while MySQL v9.2 showed better results with more result stability and resource management.

RESUMO

A segurança da informação é essencial quando os dados são partilhados entre entidades, uma vez que vulnerabilidades podem comprometer tanto o emissor como o recetor, e até mesmo sistemas inteiros. Esta tese procura avaliar o impacto no desempenho de consultas em bases de dados encriptadas. Para tal, foram identificados e selecionados algoritmos de encriptação populares e bastante utilizados, com base na disponibilidade de encriptação nativa que utilizasse os mesmos métodos e algoritmos em diferentes plataformas de base de dados. Com base nesses critérios, o algoritmo AES revelou-se o mais adequado, devido à sua grande adoção em diversas bases de dados nas variantes de 128, 192 e 256 bits.

De seguida, foram selecionadas as bases de dados que suportavam estes algoritmos, e foi concebido um ambiente de teste para cada cenário, o qual incluía diferentes algoritmos de encriptação, fatores de escala e plataformas de base de dados. Para cada cenário, foram executadas consultas a bases de dados simulando cenários reais (TPC-H). Os tempos de execução obtidos foram processados através de um script em Python, que converteu os valores de ficheiros de texto para um ficheiro de Excel devidamente formatado, associando os resultados ao número e execução da respetiva consulta de base de dados.

Os parâmetros de avaliação selecionados foram os tempos de execução e os recursos do sistema: CPU, RAM e atividade de disco. O tempo de execução foi escolhido como principal métrica de desempenho, por refletir diretamente o impacto da encriptação nos recursos de sistema no processo a ser executado. Os recursos de sistema foram analisados para garantir condições de teste semelhantes. Foi observada uma elevada variação nos resultados de desempenho em diferentes configurações, especialmente na Oracle, incluindo em alguns casos sem encriptação.

Neste estudo, foram utilizadas as bases de dados Oracle, MySQL e SQL Server para analisar o impacto do desempenho das bases de dados encriptadas em diferentes fatores de escala durante a execução de consultas. O Oracle demonstrou que o AES-192 é o algoritmo mais equilibrado em fatores de escala mais reduzidos, mas o AES-128 superou os restantes tamanhos de chave AES para um fator de escala de 10. O SQL Server apresentou uma escalabilidade de desempenho expectável, sendo o AES-192 novamente o mais equilibrado, tal como no Oracle, mas no fator de escala 1, o AES-256 teve um desempenho superior. O MySQL v8.0.38 revelou menor variação nos fatores de escala superiores, embora a estabilidade global tenha melhorado significativamente na versão Enterprise 9.2. Ao comparar o SQL Server com a Oracle, verificou-se que o SQL Server apresentou um desempenho bastante superior: a um fator de escala 10, o AES-256 foi até 40 vezes mais rápido. Mesmo sem encriptação, a Oracle foi cerca de oito vezes mais lenta do que o SQL Server. O MySQL v8.0.38 registou maior variabilidade nos dois fatores de escala testados, enquanto o MySQL v9.2 apresentou melhores resultados, com maior estabilidade e melhor gestão de recursos.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

AES – Advanced Encryption Standard

AES-NI – Advanced Encryption Standard New Instructions

CPU – Central Processing Unit

DEK – Data Encryption Key

DES – Data Encryption Standard

FHE – Fully Homomorphic Encryption

I/O – Input/Output

ISAM – Indexed Sequential Access Method

NIST – National Institute of Standards and Technology

OPE – Order-Preserving Encryption

RAM – Random Access Memory

RDBMS – Relational Database Management System

SQL – Structured Query Language

TPC – Transaction Processing Performance Council

INTRODUCTION

Data security is a priority especially nowadays, becoming a significant concern with the growing use of the Internet, particularly when it comes to the exchange of information. Studies have demonstrated that vulnerabilities in communication systems can lead to significant disruptions, including data loss, disorganization of information and compromised integrity of both organizational and personal data [1]. This has made data security a central priority for organizations that extensively rely on computer systems for business operations and data handling. One consistent risk in computer systems is the possibility for sensitive data to be accessed by unauthorized individuals. Since the risk is always present, securing database access must be a top priority [2].

Encryption serves as a technology used for providing a cryptosystem that transmits data from the source to the destination in a secret way. Various vendors have implemented many data security methods, one of which is Transparent Data Encryption (TDE), which is applied to data-at-rest with the primary goal of encrypting files and logs in real time at the disk level [3]. This thesis focuses on analyzing the performance impact of TDE on databases by measuring execution times and system resource usage – CPU, RAM, and disk activity. To measure the impact of encryption overhead, we used the TPC-H benchmark, as its dataset and queries simulate typical data-intensive operations[4].

Advanced Encryption Standard (AES) was chosen as the primary encryption algorithm due to its widespread implementation and access to native encryption in both Oracle, MySQL, and SQL Server. AES is recognized by organizations such as the National Institute of Standards and Technology (NIST) as a reliable and efficient encryption standard. AES is widely used in a variety of industries; this wide adoption reinforces its reliability and trusted status [5]. According to the official documentation, AES-256 [6] is the main TDE option supported by MySQL, while SQL Server allows the use of AES-128, AES-192, and AES-256. To evaluate the impact of encryption on performance and system resources, execution tests were conducted using multiple scale factors: 0.1 and 1 for MySQL (Community and Enterprise editions), and 0.1, 1, and 10 for both SQL Server and Oracle.

1.1 Problem Statement

Protecting sensitive data is critical today, especially with the growth of technology and concerns about data breaches and regulatory compliance. However, implementing robust encryption often brings performance overhead that can negatively affect the database performance. This creates a challenge for organizations when protecting their data and trying to avoid compromising performance.

This study addresses this challenge by evaluating the performance impact of the different Advanced Encryption Standard (AES) key lengths (128, 192, and 256 bits) in Transparent Data Encryption (TDE) on Oracle, MySQL and SQL Server databases, three of the top ranked relational databases in DB-Engines Ranking, which provides a rank of databases according to their popularity [7], [8], [9]. By comparing these encryption options, this study offers valuable insights to help guide the selection of a database platform and an encryption algorithm for securing data.

1.2 Methodological Approach

The literature review aimed at the main encryption algorithms commonly supported by different database platforms. Technical documents and articles were analyzed. The review focused on symmetric algorithms, data-at-rest encryption algorithms, and encryption algorithms implemented across different database platforms. It was searched mainly on Google Scholar, but it was found in scientific papers as well in IEEE Xplore, Scopus, DBLP, and SpringerLink. The following search keywords were used: "AES encryption in databases", "Transparent Data Encryption(TDE)", "performance evaluation of encrypted databases", "AES", "TDE", "encryption algorithms in SQL Server, MySQL and Oracle", "encrypted databases benchmark", "Oracle TDE performance", "SQL Server encryption performance", "MySQL TDE Performance", "hardware acceleration for database encryption", and "comparison of database encryption algorithms" which retrieved 167 scientific papers. After filtering and studying what could be included or not in this work, 98 articles were excluded, leaving 70 studies and official documentation fully reviewed and referenced.

Based on the literature review, the encryption algorithms selected for the study were those with native support by different database management systems. AES was considered because it had native support in many systems [10], [11], [12]. AES was selected with its key sizes of 128 bits, 192 bits and 256 bits.

To evaluate the performance, TPC-H was selected, as it is used to evaluate the performance of complex query processing and is scientifically validated in many research projects, including academic studies and industry research, bringing a set of queries to be executed against real-world business scenarios making it an excellent choice to assess system scalability and query performance.

Different scale factors were selected to assess the system's performance and database scalability. This allowed the evaluation of encryption's impact when applying different scale factors. The scale factors chosen were 0.1, 1 and 10, which are about 100 MB, 1 GB, and 10 GB, respectively. While Oracle, and SQL Server could use all scale factors, and all the AES key sizes, MySQL was limited to 0.1 and scale factor 1 because scale factor 10 took significant time and resources involved in creating the database and executing queries at scale, and was also restricted to use AES-256 only because there was no access to other key sizes.

Three different executions with different query orders, as TPC-H provides, were selected to determine the impact of encryption on database platforms. The first execution is to avoid interference with concurrent workloads or memory cache, and the following executions were necessary to assess how the database handles concurrent operations, making it crucial to understand how the database's efficiency processes queries and how it deals with resource contention. The decision to conduct three executions per test was needed to ensure consistency and accuracy in the results while also helping to avoid deadlocks while overloading the system.

With these concepts in mind, we were able to study and perform an experimental evaluation using the TPC-H benchmark and evaluate the impact of encryption on database platforms across different scale factors.

1.3 Main Contributions

The main contributions of this thesis are the following:

- Evaluation of the impact of encryption applied to Oracle, MySQL, and SQL Server databases using the algorithm AES.
- Highlighting the trade-offs between security and system performance.
- A comparison of different existing encryption algorithms.

1.4 Outline

The thesis is structured as follows. Chapter 1 introduces the motivation, the methodological approach, and the scope of the research. Chapter 2 presents the background and aims to describe the existing studies regarding selecting and evaluating AES. Chapter 3 covers related work and presents relevant prior studies, methodologies, and discussions on database encryption, as well as new key findings and understandings of the impact of using encryption in databases. Chapter 4 aims to detail the methodology and setup in which experiments are conducted to evaluate the impact of encryption on database platforms. Chapter 5 presents the experimental results obtained and their respective analysis. Chapter 6 summarizes the findings from the performance tests conducted. Finally, Chapter 7, summarise this research's main findings, address its limitations and suggest potential directions for future work.

2 BACKGROUND

This chapter begins with DES and TDES, foundational encryption standards, to provide the necessary context and historical perspective before explaining AES. DES and TDES are presented first to understand the advancements that AES brought to the field of encryption. Thereafter, the chapter explores AES in detail, describing the existing studies regarding selecting and evaluating AES, one of the most widely used and respected encryption methods. The AES algorithm was chosen because of its widespread adoption and use. According to the literature in [13], [14], [15], AES is one of the most used encryption algorithms and provides robust cryptographic properties, being a fundamental component in security application.

2.1 DES

DES was developed in 1972 and adopted in 1977 as Federal Information Processing Standard (FIPS) by the National Bureau of Standard (NBS) [16].

DES starts with a 64-bit block plaintext and returns a ciphertext of 64-bits as well. During the encryption, the algorithm uses these steps: permutation at the beginning and at the end of encryption; 16 rounds of Key transformation, Expansion Permutation, S-Box, P-Box or permutation box, XOR right half operations output value with left half value, and Swap Left half with Right half [17].

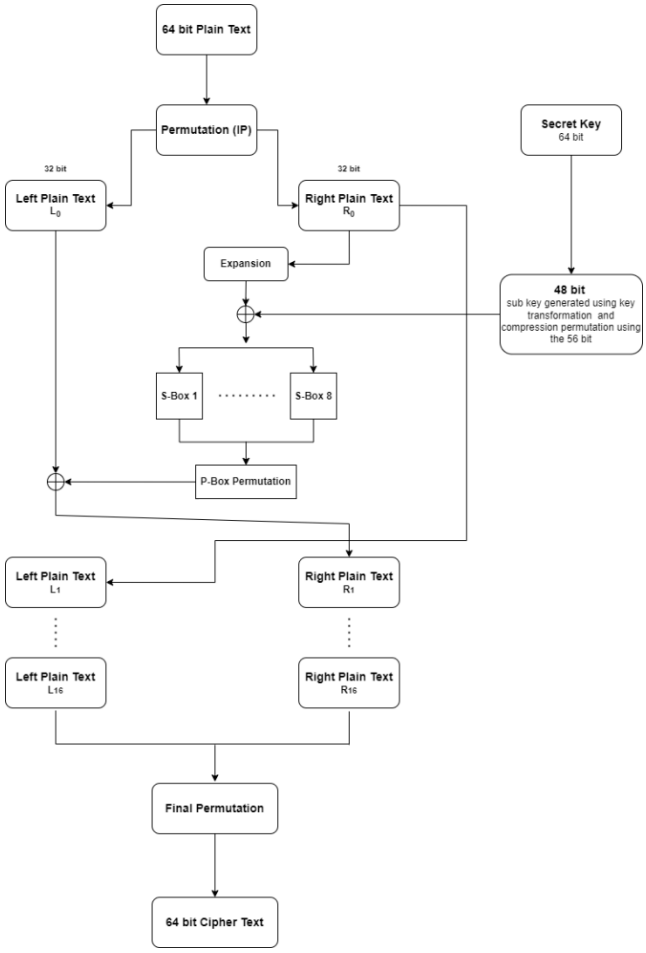


Figure 1 - DES Encryption Flowchart (source: [15], [16])

Figure 1 introduces a big picture of the algorithm flow in terms of encryption. Before starting the rounds, there is an Initial Permutation (IP) where each bit of the plaintext will change positions. After the permutation, the 64-bits of plaintext divide into two blocks of 32-bits called Lo and Ro. From this moment, the 16 rounds of encryption begin. Ro is the most used side in each round, and at the end, the Ro and Lo are XOR and swap places.

So, as the first step of a round is the Key transformation, the key size for the Data encryption standard is a 64-bit key length, which is reduced to a 56-bit key, removing every 8^{th} bit from the 64-bit key. In each round, from the 56-bit key available, it generates a 48-bit subkey; the way to generate this 48-bit subkey is by first dividing the 56-bit key into two halves of 28-bits, then there is a shift, depending on which round, of 1 or 2 bits to the left between the two blocks: on rounds 1, 2, 9, and 16 the bits are rotated to the left by one position, and on rounds 3, 4, 5, 6, 7, 8, 10, 11, 12, 13, 14, and 15 are rotated to the left by two positions [18].

After the key transformation, a 48-bit sub-key is selected from the 56-bits from the key transformation using compression permutation [19]. The Key transformation will occur every round, and from the 56-bit key, it creates a "new" 48-bit subkey.

The next step is Expansion, the 32 bits from the right side block expand to 48-bits by dividing the 32-bits into eight blocks of 4 bits, and each block of 4 bits transforms into a block of 6 bits by passing the right significant bit of the block i of 4-bits to the first position of the block i+1 and the leftmost significant bit of the block i+1 to the last position of the block i. Now the output from the Key transformation is subject to an XOR operation with the Expansion Permutation [18] that will it is used in the Substitution Boxes step.

In each S-Box, the input is 6-bit; the output of each sub-block will be 4 bits which returns a 32-bit block when joining the 8 S-Boxes. The S-box takes the 6-bit and the most significant bit of the right and left side to find the row on the S-Box table, and the four inner bits of the 6-bits to find the column; once the row and column values are known is possible to find the 4-bit output in the S-Box table [20].

Now, as the last step, the output from the S-Box is first permuted in a P-Box permutation, and the result of the permutation is XOR with the Lo side, and it becomes the new Ro side value of 32-bits, and the Ro initial value becomes the Lo input for the next round.

Once the 16 rounds finish before giving the plaintext encrypted, it is subject again to a final permutation, and the result is the 64-bit ciphertext.

According to Alanazi et al. [21], DES is considered obsolete because it can be cracked using brute force. In 1998, the Electronic Frontier Foundation built a DES cracker that could decode DES messages in less than one week. T-DES, described in the 3.1.2 section, is a version of DES created to protect against brute force attacks connected to weak keys [22]. According to Hercigonja [23], "The weak S-boxes provide a possible mean for a cryptanalytic attack. DES is highly susceptible to linear cryptanalysis attacks. It is exposed to brute force attack because of the weak keys."

2.2 T-DES

T-DES or 3DES, referred to as Triple Data Encryption Algorithm (TDEA), was proposed by IBM in 1988. T-DES comes from DES, but as DES security is weak, T-DES comes to improve DES's algorithm without needing to build a whole new algorithm to encrypt data.

The difference between T-DES to DES is the key size. Instead of having the 56-bit key, which now has three times the size, 168 bits (3 times 56 bits), it can be 112 bits and 168 bits of key length depending on the number of keys, two sub-keys, and three sub-key, respectively, having 48 rounds and brute force now it is not a threat to this algorithm as this algorithm is present in several Internet protocols. The only disadvantage of DES is that it takes more time to encrypt [24].

Triple Des Encryption will be three times the DES encryption, as shown in Figure 2. It can use three sub-key or two sub-key, e.g., 112-bits or 168-bits of key length. For example, for the T-DES with 3 Keys, the plaintext will be encrypted using K1 (First key set) and will give as output the C1 ciphertext, next step, C1 will be encrypted using the K2 (Second Key set) and will give the C2 ciphertext, on the last step the last sub-key will encrypt the C2 and retrieve the final ciphertext [16].

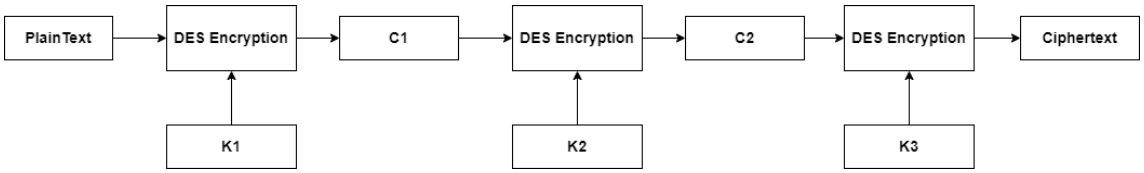


Figure 2 - T-DES Encryption Flowchart (source: [25])

In Figure 3, the process will be the same for the decryption, using the three keys to decrypt each ciphertext until the last step, where the result is the plaintext.

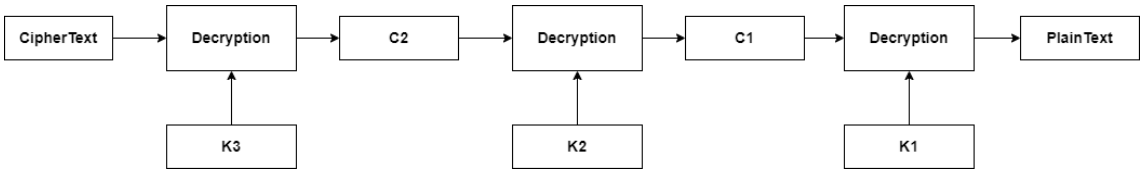


Figure 3 - T-DES Decryption Flowchart (source: [25])

For the triple Des with two keys, the process will be the same, but instead of being K1, K2, K3 will be K1, K2, K1 and decrypt in the same order.

2.3 AES

This section aims to explore the AES fundamentals, providing a breakdown of the structure with a step-by-step description of each encryption process stage.

There was a need to substitute the DES and 3DES in 1997 to create an algorithm with better security and efficiency, and AES was the next in line. Rijndael finished its development in 2001 with Vincent Rijmen and Joan Daeman [16].

AES is a symmetric key, meaning the sender and receiver will use the same key on the Symmetric Key to encrypt and decrypt. The Symmetric Key, also known as Secret Key Encryption, is easy-of-use but lacks security where the key is shared between the sender and receiver. The message can be decrypted if someone intercepts the key [26].

The Key length can be 128 bits, 192 bits, and 256 bits, and the number of rounds will depend on the number of bits. If the key length is 128 bits, it will have ten rounds; if it is 192 bits, it will be twelve rounds; and if it is 256 bits, the number of

rounds is fourteen. All sub-processes in the encryption process use a matrix of 4x4, and instead of working with bits, AES works with bytes so that 128 bits will be 16 bytes in a matrix of 4x4. AES has four steps, except in the last round, where the mix columns are unnecessary [5], [27], as stated in Figure 4.

The plaintext of 128 bits transforms to a new state array by XOR, the 16-byte matrix with K_0. After getting the plaintext in a new state, it uses the new state array of the plaintext in the first round. The first sub-process is substitution bytes that will use a 16x16 matrix of byte values called S-Box defined by AES, which will be very useful to convert the plaintext transformed state array to a new state array [27].

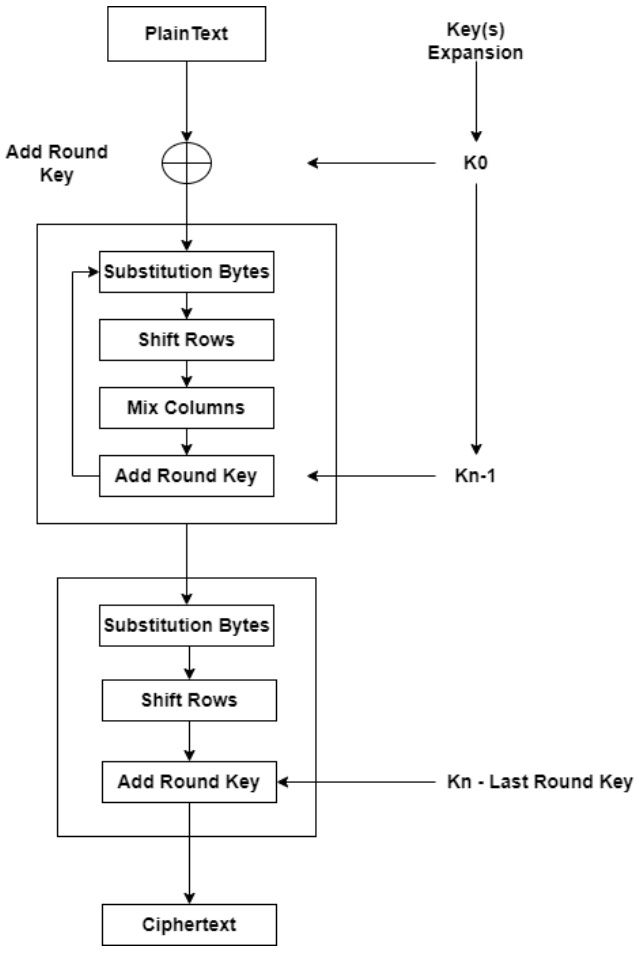


Figure 4 - AES Encryption Flowchart (source: [5])

Figure 5 describes the Substitution byte function that will convert byte by byte. For example, to convert the first byte, it is necessary to understand that the leftmost 4 bits will be the row in the 16x16 matrix, and the four rightmost bits will be the column. So, the byte converts to the one located using the row and column indexes originating from the byte in the state array. The process will be the same for the rest of the bytes in the state array [5], [27].

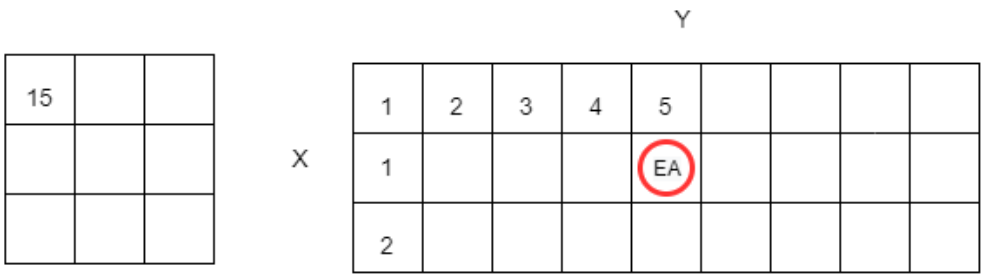


Figure 5 - Substitution Bytes (source: [5], [28])

The next sub-process is Shift Rows; as the name says, it will shift the bytes for every row and will add more complexity to the algorithm; on each row, the process of shifting is different; the first row will stay the same, and the second row will shift one place to the left, the third row will shift two places to the left and the last row will shift three places to the left, the number of places to be shift depends on the block length [29].

After shift rows, are the mix columns, the mix columns will use a transformation matrix to transform the state array. The operation performed is multiplying a column by a given matrix for each column.

The last step of each round is the Add Round key, an essential step, and it is the same as the first before the round, the initial transformation of the state array. The columns from the key matrix are bitwise XOR with the columns of the state array which is going to create a new ciphered state array.

AES Decryption, Figure 6, will be in reverse order. The last round will always have the same three steps as Substitution Bytes, Shift Rows, and Add Round key [5].

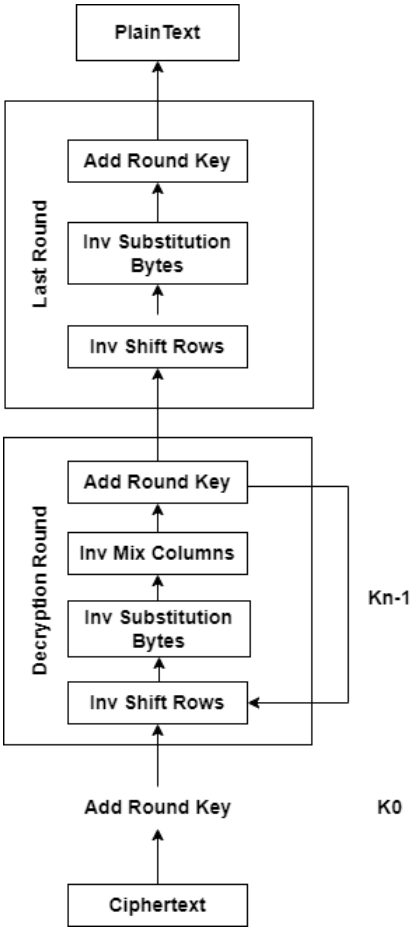


Figure 6 - AES Encryption Flowchart (source: [5])

2.4 Transparent Data Encryption (TDE)

Transparent data encryption is a security feature to ensure data-at-rest security in database platforms. TDE works automatically when enabling TDE encryption, encrypting the data stored in the database files, such as tablespaces or data files, without requiring any function to decrypt the information when accessing or any function to encrypt when inserting. This process is called "transparent" because it does not require any changes to SQL queries or database schema, making the user experience seamless for users and developers. This encryption guarantees that the data files will remain protected if they are stolen or accessed [3]. TDE is present in many database platforms, offering different encryption algorithms to be used with TDE.

TDE was introduced by different DBMS to address the requirements associated with public and private privacy and security mandates such as PCI [30]. TDE was built in different DBMS individually as each of the vendors created their transparent data encryption separately as a feature. Oracle introduced TDE in Oracle Database 10g Release 2, focusing on column level and later expanded to tablespace level encryption in Oracle Database 11g Release 1 [30]. Microsoft incorporated TDE into SQL Server 2008, providing full database capabilities, Microsoft has not publicly

given credits for the creation of TDE to specific individuals, its development was part of new developments in SQL Server 2008 to improve data security and compliance [31], [32]. MySQL transparent data encryption version was introduced as part of the MySQL Enterprise Edition in version 5.7.12 by MySQL team at Oracle [33].

TDE is enabled on the database, the encryption and decryption process happen automatically and transparently at the storage level. When data is written to the disk, it is encrypted using a symmetric key. When data is read by an authorized user or application, it is decrypted by the database engine without requiring any functions to decrypt the information at query level, stored procedures, or application. At SQL Server it's possible to enable and disable the transparent data encryption [31], in MySQL the tablespace itself is encrypted, everything stored in the tablespace will be encrypted not allowing to disable and enable [34], and in Oracle the tablespace is encrypted and it is possible to rotate the algorithm encryption on the tablespace, in this case it's possible to change AES among the available key sizes and once a table is moved inside the encrypted tablespace the table is automatically encrypted [35]. In Figures 7, 8, and 9, for Oracle, MySQL, and SQL Server respectively is possible to see their own TDE mechanisms. Each figure details how TDE is enabled, and the specific steps followed to configure the encryption in the respective database systems.

TDE is used because over cell-level encryption, its ease of implementation using a "flip-the-switch" solution which doesn't require functions to encrypt and decrypt, minimal performance overhead, it only introduces a small overhead compared to cell-level encryption, being transparent makes it more light in terms of operation offering security and efficiency, TDE also helps organizations to meet data protection requirements across different regulations (GDPR, HIPAA, PCI DSS), and unlike column level encryption TDE encrypts all data files, including backups and log files, providing end to end protection for sensitive data [30], [32], [33]. Because of these advantages over TDE in terms of efficiency between security and small overhead compared to cell-level encryption, TDE was chosen to be the case study of this thesis.

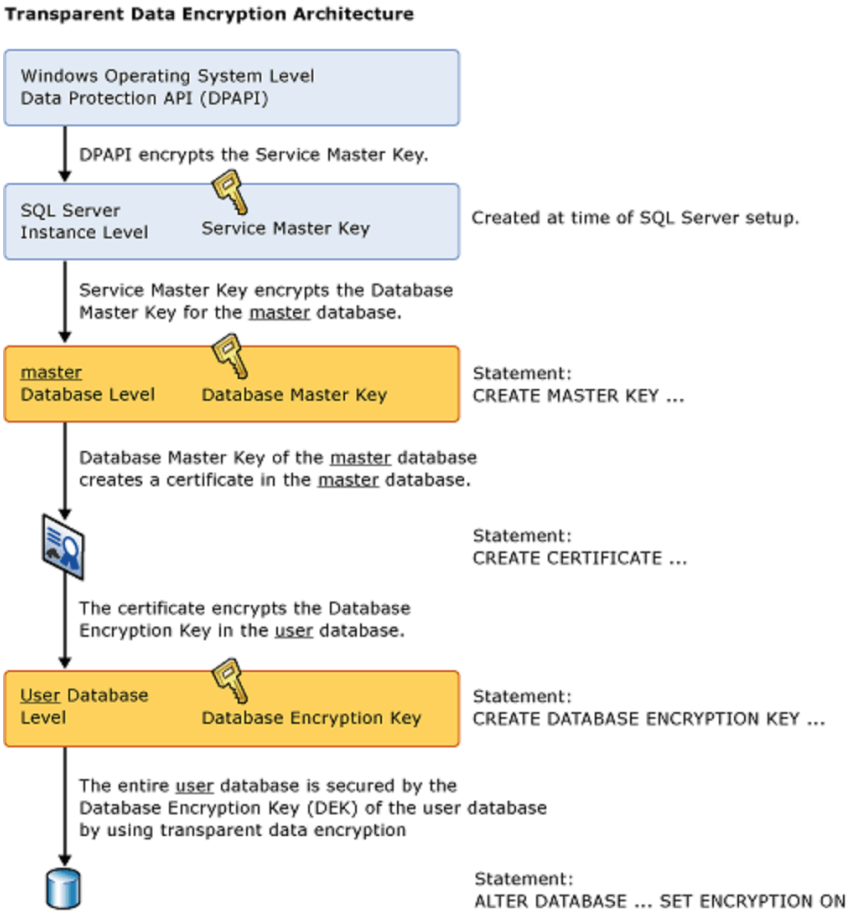


Figure 7 – Microsoft TDE encryption architecture (source: [31])

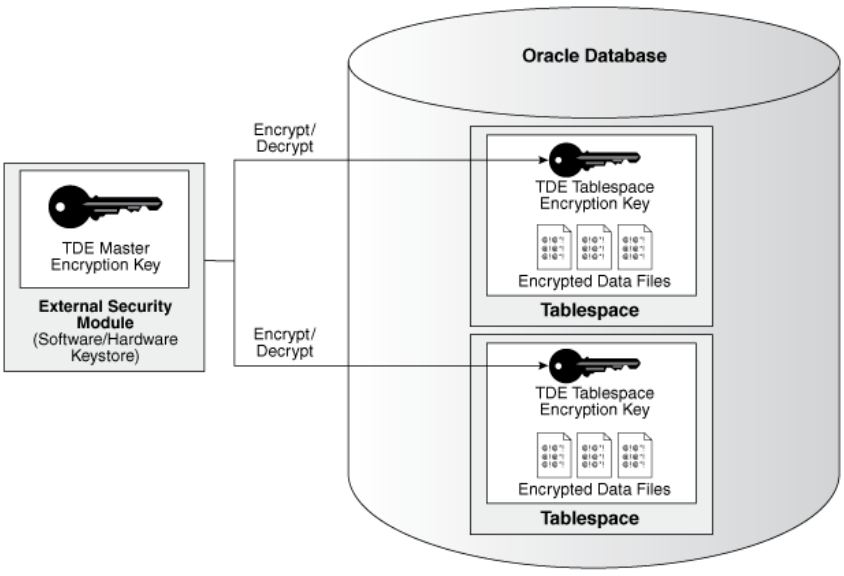


Figure 8 - Oracle TDE Tablespace Encryption (source: [35])

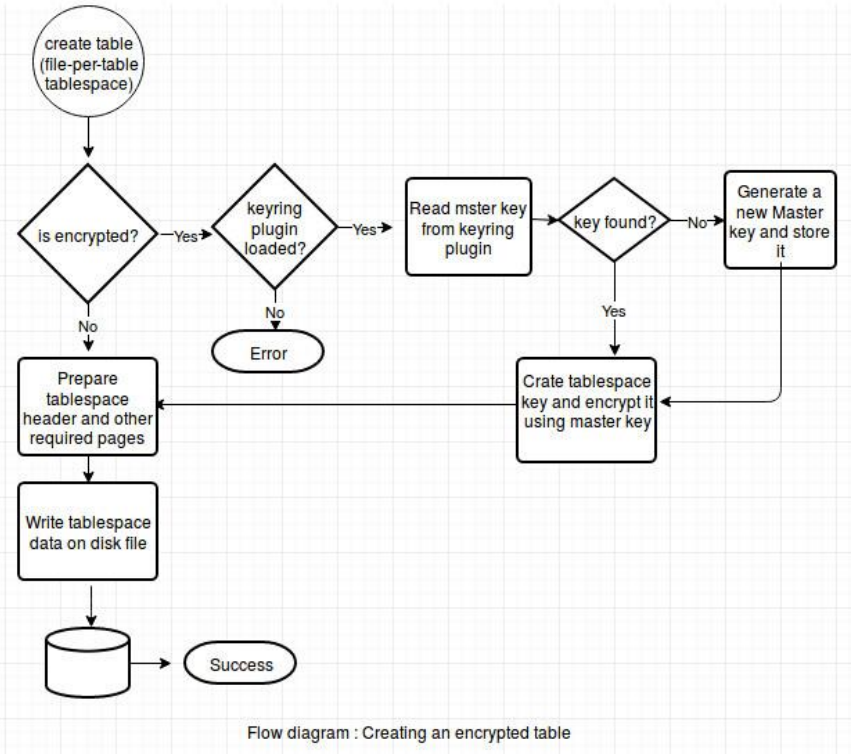


Figure 9 - InnoDB Transparent Tablespace Encryption (source: [36])

2.5 Encryption algorithms in different database platforms

Table 1 offers an overview of encryption algorithms used in different database platforms. It highlights the encryption algorithms that are available on each platform and shows that AES is the most natively supported encryption algorithm across different databases. The databases selected were Oracle, MySQL, and SQL Server due to the support to TDE and for being recognized databases across different industries and academic research.

Table 1, created by the author, summarises well-known databases and evaluates the performance between different platforms, focusing on encryption type, data-at-rest support, and supported algorithms. Most of the platforms are compatible; platforms like Oracle, MySQL, and SQL Server support TDE and use AES as an encryption algorithm; however, PostgreSQL does not support TDE, and it relies on third-party solutions. The Database versions inserted were the ones used for this project and in the next chapters the databases mentioned will be related to those versions inserted in the table, except for MySQL because it was done a comparison work between two distinct versions, one being MySQL Version 8.0.38 Community Server; and another being MySQL Version 9.2.0 Enterprise Server.

Table 1 - Comparative Table of Encryption Algorithms for Different Database Platforms

Database Platform	**Encryption Type**	**Encryption at Rest**	**Supported Algorithms**	**References**
MySQL (MySQL Version 9.3.0 Community Server)	Transparent Data Encryption (TDE), AES_ENCRYPT(), AES_DECRYPT() (manual column-level encryption)	Yes (AES-256 for TDE)	AES-128, AES-192, AES-256 (column-level), AES-256 (TDE)	[34], [37]
PostgreSQL (Version 17.4)	pgcrypto extension (client-side encryption)	No (requires third-party tools)	AES, DES, Blowfish	[38], [39]
SQL Server (Version 2022 Build 16.0.4185.3)	Transparent Data Encryption (TDE), Always Encrypted, Column-level encryption	Yes	AES (128, 192, 256-bit), RSA, Triple DES (128, 192-bit)	[31], [40], [41]
Oracle Database (Version 23ai)	Transparent Data Encryption (TDE), Column-Level Encryption	Yes	AES (128, 192, 256-bit), 3DES	[42]
MariaDB (Version 13.0)	Transparent Data Encryption (TDE), Column-Level Encryption	Yes	AES-256, RSA	[43]

3 RELATED WORK

To find relevant papers, keywords were used to start the search. The keywords were "AES encryption in databases", "Transparent Data Encryption(TDE)", "database performance with encryption", "AES", "TDE", "encryption algorithms in SQL Server, MySQL and Oracle", and "performance SQL encryption", "encrypted databases benchmark", "Oracle TDE performance", "SQL Server encryption performance", "MySQL TDE Performance", "hardware acceleration for database encryption", and "comparison of database encryption algorithms". Scientific database sources and platforms such as Google Scholar, DBLP, Research Gate, and IEEE Explore were used to search for technical documents that could be used to search for related work. The selection criteria focused on documents discussing transparent data encryption, its practical applications, and performance studies involving Oracle, MySQL and SQL Server, emphasising the most recent and relevant studies available. Out of 55 fully reviewed studies, 19 were selected as the most relevant to be included in the Related Work section.

This chapter aims to describe the existing studies regarding evaluating the Impact of AES Encryption on Query Read Performance Across Oracle, MySQL and SQL Server Databases and provide insights into relevant studies related to the work. Data security will always be important in protecting everyone's sensitive data. There will be a need to study and improve the encryption algorithms available to provide this layer of protection to data.

In master thesis [11], it is developed and validated a benchmarking framework designed to evaluate the performance impact of Transparent Data Encryption on database operations. This works uses metrics such as elapsed time, CPU time, throughput, and storage impact. Tested on SQL Server 2019 and MySQL 8.0 with different key lengths and algorithms (AES and 3DES), revealing that TDE shows a modest overhead, most of them in insert operations on MySQL and that AES outperforms 3DES given the hardware-level acceleration. Its findings confirm that there are different performance impacts between vendors. Overall, Microsoft SQL Server 2019 outperformed MySQL in terms of elapsed time, throughput, and CPU efficiency when Transparent Data Encryption (TDE) was enabled.

The work [44] evaluates the performance impact of using encryption in DW, focusing on Oracle TDE with AES and 3DES algorithms. Using the TPC-H benchmark at 1 GB and 10 GB scale factors, it was possible to find that the response time overhead ranges from 30% to 163% and the individual query time overhead ranges from 100% to 1000%, when Oracle claims that the expected results of response time overhead are, on average, between 5% and 10%.

The work in [3] is to see if the Transparent Data Encryption system and Non-Transparent Data Encryption show any differences in system performance regarding hardware usage. It was discovered that either one system or another don't have a significant difference when compared to CPU usage and memory consumption, with

TDE having an increase of 8% in CPU usage and 2% to 4% increase in memory usage. Additionally, a backup process study showed that TDE added an overhead of approximately 10% to 18% during backup operations.

In [45], the author focus on edge computing and cloud systems where the goal is to bring the data to a cloud system, avoiding the increasing and limited data storage that edge computing has, calling it Cloud/Edge computing. To avoid risks to the data being transferred from the edge system to the cloud system, the use of Transparent Data Encryption is proposed, creating a solution to encrypt the data locally at the edge and transfer them to the distributed database over the Cloud. This study proves the extensibility of TDE in different scenarios and applications in which security and privacy are crucial.

The work done in [10] focused on MySQL and MariaDB, in this project it was done an Analysis of Variance (ANOVA) test comparing MariaDB's transaction throughput under AES-128, AES-192, and AES-256, which gave a result of p-value of 0.5781, which means that there is no significant difference in throughput among the three key lengths. The mean of transactions per minute (TPM) gave 268261 for the unencrypted database, 275717 for AES-128, 274798 for AES-192, and 274501 for AES-256, showing that there is no impact, or a high overhead compared to the results without encryption. When doing performance tests to MySQL, by contrast, MySQL's ANOVA test produced a p-value of 0.0002915, which means there's differences among the results, and in terms of mean TPM produced a value of 103445 to no-encryption, 103730 to AES-128, 104371 to AES-192, and 102499 to AES-256, showing a drop in AES256. MariaDB a slight overhead without a measurable slowdown, where MySQL showed a small but statically slowdown hit in AES256 and overall MariaDB processes transactions around double MySQL's rates.

In [46], the author focuses on doing a comparative analysis of Oracle, MySQL, and SQL Server databases. The findings show that all three databases are well-suited for analytical and data science queries. However, Oracle and SQL Server showed to be the most efficient databases when running group functions over large sets of data than MySQL, being only proficient in handling simple aggregations. In addition, Oracle and SQL Server offer more robust security than MySQL. They are better choices for systems that require more complex data processing demanding security, scaling, and integration with non-relational or unstructured data. MySQL can be used for lighter applications that do not require advanced features.

In [47], explains the AES-IN functionalities which implement complex parts of the AES algorithm, such as SubBytes, Shiftrows, Mix Columns, and AddRoundKey transformations, directly in the hardware, it reduces the instruction count for AES operations. The results show that AES-NI achieves up to 13.5x speed over AES traditional at 90% reduced energy consumption over AES. AES traditional refers to using the AES algorithm without hardware-accelerated AES-NI instructions.

The work performed in [12] presents a preliminary assessment of the protection offered by Oracle, MySQL, and PostgreSQL. It investigates encryption-at-rest and

transparent data encryption, offering recommendations for a preliminary actionable plan of use. MySQL supports data-at-rest and utilizes the AES-256 encryption algorithm. Postgresql does not support built-in TDE like MySQL; only using EnterpriseDB is possible to have TDE. Instead, users can use plugins and solutions, such as pgsql-hackers TDE, that might lead to additional work as the user is responsible for maintaining ownership.

The study [48], focuses on enhancing query performance and privacy in relational cloud databases. This study introduces SecureSQL, a framework that combines AES-128 encryption with hash map indexing to optimize query execution on encrypted data. SecureSQL uses 20 of the 22 TPC-H benchmark queries efficiently and it outperforms conventional encrypted databases by reducing execution time without modifying the DBMS structure. The study enhances privacy-preserving database outsourcing while ensuring fast and scalable query execution for real-world applications.

The study [49] conducted a comparative analysis of Oracle and MySQL databases, specifically measuring query execution times and scalability under TPC-H workloads using Apache Jmeter to simulate transactional workloads. The study found that Oracle has lower and more consistent response times compared to MySQL that shows a high variability response time. MySQL while effective in lightweight and read-heavy scenarios didn't show consistency as Oracle showed, being Oracle considered to be more suitable for high- throughput and critical environments.

Katsadouris's master thesis [50] examines encryptions for secure databases, comparing OPE, FHE, and searchable encryption and their implications for efficiency and execution of queries. The work examines secure designs for databases, including CryptDB, and mentions efficiency of trade-offs for encrypting queries. Results demonstrate the impact of encryption for storing, retrieving, and indexing data, where efficiency and security need to be balanced. This dissertation uses TPC-H benchmarks to assess how encryption affects query performance in practical scenarios.

Ansmink's master thesis [51] studies encrypted query processing in DuckDB by analysing how encryption affects query execution time and storage efficiency and system performance. The research integrates Intel SGX with column-level encryption in DuckDB's architecture which produces an average 22% performance overhead. The system achieves a maximum compression benefit of 2.12× by implementing compression techniques. The study shows that stronger encryption can impact performance and storage requirements.

The study [52] proposes an improved Transparent Data Encryption mechanism for cloud-based DBMS securing data in use and at rest. It improves security by encrypting active data during query execution. The difference between this approach and other encryption methods is that it allows normal SQL queries without modifications, ensuring usability which is one of the goals of this study. Findings show that

data-in-use encryption introduces minimal performance overhead, making it a solution to improve security and usability.

The main purpose of research [53] is to study Transparent Data Encryption (TDE) in Oracle databases (versions 19c, 18c, 12c) and how it may influence the performance of a CPU, RAM, and I/O. The results of the study show that TDE uses nine times the amount of RAM compared to non-encrypted database configurations. In terms of percentage, non-tde databases used between 10% and 16% of RAM, while TDE databases used between 83% and 90%, regardless of the Oracle version. The results show a clear trade-off between security and system efficiency, which is important when deploying TDE in production environments.

The work [54], applied an experimental benchmark using application-layer AES-256 encryption via Hibernate and Jasypt, migrating an e-commerce application (B2B marketplace) to the cloud. It measures the performance overhead of database transactions (insert, read, update, and delete), revealing significant penalties. Insert operations add a 4.5-fold increase in execution time, while read, update, and delete operations increased 2 to 2.85 times. The main contributions include detailed evidence of TDE performance impact, guidance when selecting critical columns to minimize the overhead and recommendations for adjusting service-level agreements (SLAs) and quality-of-service (QoS) parameters according to the cloud environments.

The study [55], acknowledges TDE as used database-level encryption method, suited for protecting data at rest by encrypting and decrypting data transparently within the DBMS. However, there's some limitations highlighted, TDE provides no protection against application-level attacks, since encryption occurs only at storage layer and decrypted data is exposed once accessed by the database engine. The paper also adds that TDE lacks support on accelerated index-search on encrypted columns, making it not optimal when query performance is required. These limitations are faced with hybrid solutions that combine software level encryption with hardware-based key protection which the author finds more flexible and more performant.

There is some work done that uses transparent data encryption features, in [56], the authors analyzed five database encryption architectures and identified the trade-offs between performance, transparency, and security. The authors proposed a novel "above-cache" which takes place at the encryption module inside DBMS, just above the cache layer. This approach allows cell-level encryption with minimal performance overhead and transparency to the application layer. It was implemented and tested four different encryption models, including this new approach and when comparing to each other it was found that above-cache model outperforms others in realistic cache scenarios. The study proves that strong security and performance can coexist through strategic DBMS-layer integration.

Another relevant study [57] provides performance evaluation results between SQL Server's Transparent Data Encryption (TDE) and NetLib Encryptionizer which both use AES-128 encryption. The performance evaluation included five core operations: retrieval, insertion, update, deletion and backup across encrypted and

unencrypted instances. The test occurred across three different environments which included virtual pc and VMware and a non-virtual dual core machine. The results demonstrated that Encryptionizer provides better performance than TDE because it delivered 17.2% faster execution in non-virtual environments. The performance of TDE resulted in an average 25.2% decrease compared to an unencrypted system. The benchmark standard TPC-C or TPC-H was not used because the research employed a custom AS3AP-based benchmark which used a balanced methodology to reduce system noise and cache interference effects. This work was also important to reinforce the performance cost of database-level encryption and the importance of tool selection in enterprise environments.

A recent study [58] presents a performance comparison between Oracle Database Server and Microsoft SQL Server with particular emphasis on encryption performance. The research was on AES-128, AES-192 and AES-256 execution times on both platforms for a 68910124 bytes data set. The study showed that SQL Server has better execution on encryption and decryption operations than Oracle Database Server, when using AES-128. The study showed that the encryption algorithm selection resulted in performance overheads of approximately 22% for Oracle and 28% for SQL Server.

This study focuses on performing a direct comparative analysis of AES encryption performance across Oracle, MySQL and SQL Server using TPC-H benchmarks to simulate realistic query loads with different key lengths.

In Table 2, presents the key findings from the related works mentioned earlier. It lists the databases, algorithms used, main findings, and citation count. The table highlights how encryption impacts performance across different DBMS.

Table 2 - Related Work: Encryption Algorithms, Databases, and Key Findings

Ref.	Databases	Algorithms	Key Findings	Number of Citations
[11]	Microsoft SQL Server 2019, MySQL 8.0	AES (128, 192, 256 bits), 3DES (only in SQL Server)	• SQL Server outperforms MySQL overall. • AES shows significantly better performance than 3DES in both elapsed and CPU time, thanks to hardware acceleration. • TDE introduces minor overhead, preserving its "transparent" nature. MySQL struggles more with DELETE and INSERT operations,	1

Ref.	Databases	Algorithms	Key Findings	Number of Citations
			showing higher latency and CPU usage.	
[44]	Oracle 11g, Microsoft SQL Server 2008	SES-DW (custom cipher using XOR + MOD), AES (128/256), 3DES, OPES, Salsa20	• The SES-DW encryption solution provides superior performance for numerical data stored in data warehouses compared to standard encryption algorithms (AES, 3DES). • The evaluation considers storage requirements, loading time, and query response time. • The solution preserves original datatypes while adding minimal overhead. The solution provides better scalability and maintains a good security balance.	10
[3]	Microsoft SQL Server 2017	TDE (AES, SQL Server native)	• The implementation of TDE results in an 8% CPU increase and between 2% and 4% memory usage. • Backup operations take between 10% and 18% longer. • The transaction rate experiences a maximum reduction of 7%. • The performance impact remains moderate and easy to forecast. The security solution provides practical protection without significant performance drawbacks.	13
[45]	Cassandra, MongoDB, and CouchDB	AES-256 (custom symmetric TDE)	• The Custom TDE system exists specifically for CouchDB operations	3

Ref.	Databases	Algorithms	Key Findings	Number of Citations
			within Cloud/Edge environments. • The system encrypts data at the Edge level before storing it in the Cloud. • The system uses a local index to find encrypted fields during query operations. The TDE-enabled CouchDB system provides complete decentralization together with privacy features and resilience capabilities.	
[10]	MySQL 8.0.29, MariaDB 10.7	AES-128, AES-192, AES-256 (data-at-rest encryption via InnoDB)	• AES-256 performance decreases dramatically compared to AES-128/192 encryption when used in MySQL. • The AES key lengths results show no significant differences in MariaDB. • The throughput of MariaDB reached two times higher than MySQL. AES-256 remains the recommended encryption standard when security requirements exceed performance needs.	2
[46]	Oracle 19c, SQL Server 2019, MySQL 8.0	AES (used for data-at-rest encryption via TDE SSL/TLS (for data-in-transit encryption).	• Oracle and SQL Server deal better with complex queries than MySQL. Oracle excel in indexing capabilities. Oracle excel in security features. MySQL delivers good performance for standard operations.	2

Ref.	Databases	Algorithms	Key Findings	Number of Citations
[47]	Not tested within a DBMS	AES-256 (CBC mode) with and without AES-NI hardware acceleration	• AES-NI provides performance that exceeds traditional AES by 13.5 times. • The technology cuts power usage by 90% which makes it suitable for IoT and edge devices. • The system delivers exceptional performance in confusion, diffusion and randomness capabilities. The system fulfils all requirements of the Dieharder and NIST randomness suites testing protocols.	20
[12]	MySQL (InnoDB), PostgreSQL (pgsql-hackers, Cybertech, EnterpriseDB), Oracle 19c	AES-128, AES-192, AES-256, with TDE, 1-tier & 2-tier models, key rotation, and external key management options	• MySQL supports TDE. • PostgreSQL does not natively support TDE. • MySQL and Oracle support built-in TDE with minimal overhead. • Oracle showed the best performance with parallel loading. PostgreSQL had the slowest encryption process.	-
[48]	MySQL (hosted on Google Cloud SQL)	AES-128 in CBC mode, Client-side encryption with hash map indexing	• Introduces SecureSQL, a client-side encrypted query framework. • Uses AES-128 (CBC) and a hash map for efficient encrypted lookups. • Executes 20 of 22 TPC-H queries with better performance than traditional methods. No DBMS change needed; shows feasibility of secure, efficient outsourcing.	-

Ref.	Databases	Algorithms	Key Findings	Number of Citations
[49]	Oracle 19c, MySQL	No-encryption	• The JMeter performance tests demonstrated that Oracle achieved better response times, which remained stable throughout the tests. • The system demonstrates superior performance for high-throughput critical workloads because it provides robustness and scalability. • MySQL demonstrates strong performance in scenarios that involve heavy reading operations and lightweight transactional operations. The selection depends on workload type and cost, with required features.	-
[50]	CryptDB	OPE, FHE, Searchable Encryption, Homomorphic Encryption, Property-Preserving Encryption (PPE)	• Encryption affects query execution performance, depending on the type of encryption used. • Efficiency and security trade-offs must be considered when choosing encryption for real-world databases. Stronger encryption schemes (FHE) offer better security but introduce significant computational overhead.	-
[51]	DuckDB	AES-based Authenticated Encryption (e.g., AES-	• The implementation of encryption in DuckDB results in approximately 22% performance degradation when running TPC-H queries.	4

Ref.	Databases	Algorithms	Key Findings	Number of Citations
		GCM), Intel SGX	• The application of compression methods leads to performance improvements reaching up to 2.12 times. • The vectorized engine of DuckDB supports column-level encryption with high efficiency. The performance of Intel SGX depends on workload complexity.	
[52]	Cloud DBMS	Homomorphic Encryption (ADD, MUL), Order-Preserving Encryption (OPE), Deterministic Encryption (DET)	• Proposes a transparent encryption model that secures data-at-rest, data-in-use, and partially data-in-motion. • Introduces cryptosets (multi-layered encrypted representations of data). • Supports atomic and isolated execution of complex SQL queries, including joins and transactions. Achieves security without exposing encryption keys or plaintext to the untrusted cloud DBMS.	2
[53]	Oracle 12c, 18c, 19c	AES-256 (Oracle TDE default)	• TDE significantly increases RAM usage, consuming up to 9× more memory than non-encrypted configurations. • Oracle 19c had an improved I/O performance compared to 12c and 18c. With TDE the CPU usage and execution time are slightly higher, but still functional.	9

Ref.	Databases	Algorithms	Key Findings	Number of Citations
[54]	PostgreSQL (with Hibernate/Jasypt middleware encryption)	AES-256 (CBC mode) via Java Cryptography (JCE) and Bouncy Castle	• Middleware-level AES-256 encryption introduces moderate to significant performance overhead during database operations. • Selective encryption of critical data fields is recommended to reduce unnecessary performance cost. • Insert operations are the most impacted, with up to 4.5× increase in execution time. Read, update, and delete are slowed by 2.0 to 2.85× due to encryption.	55
[55]	Oracle, SQL Server, DB2 (used in benchmarks); general RDBMS focus	AES, RSA, DES, Blowfish (AES preferred); software + HSM + network-attached encryption	• TDE protects data at rest but offers no protection at the application layer once data is decrypted. • TDE does not support efficient indexing or search over encrypted columns, impacting query performance. • Hybrid models (sw encryption + hw key protection) provide a better balance between security and performance. Selective, column-level encryption is essential to reduce overhead while securing sensitive data fields.	32
[56]	MySQL (with custom implementation)	AES-128 (CBC mode), SHA-1 (for cell binding)	• Evaluates five database encryption architectures and their impact on performance, security, and transparency.	54

Ref.	Databases	Algorithms	Key Findings	Number of Citations
			• Supports cell-level encryption with minimal performance overhead. Maintains full transparency to the application laye Outperforms all other architectures under realistic cache constraints, proving security and performance can coexist.	
[57]	Microsoft SQL Server 2008	AES-128 (TDE and NetLib Encryptionizer)	• Compared SQL Server TDE and NetLib Encryptionizer, both using AES-128 encryption. • Encryptionizer outperformed TDE, with up to 17.2% faster execution in non-virtual environments. TDE show a 25.2% performance drop compared to unencrypted systems.	-
[58]	Oracle 19c, Microsoft SQL Server 2022	AES (128/192/256), RSA, 3DES, DES, Blowfish (Oracle), ChaCha20 (SQL Server)	• SQL Server showed faster encryption and decryption performance, especially with AES-128. Performance overhead due to encryption was approximately 22% for Oracle and 28% for SQL Server.	1

4 EXPERIMENTAL SETUP AND METHODOLOGY

This chapter aims to detail the setup before the experiments are conducted to evaluate the impact of encryption on database platforms. This includes a description of the operating systems used, the database platform versions, the benchmark used to allow for a standardized evaluation, the data upload process, and the way data-at-rest encryption was applied.

4.1 TPC-H

The benchmark TPC-H was selected to evaluate the performance of the algorithms being executed on the different database platforms. The TPC-H benchmark is one of the standards to be used. It is quite popular when comparing databases and query performance [59]. This work used the TPC-H to measure the queries performance on the selected databases. Using different scale factors (SF), we per-form our performance evaluation using 100 MB (SF=0.1), 1 GB (SF=1), and 10 GB (SF=10) datasets to observe scalability effects [60]. We executed all TPC-H queries to evaluate the performance of the selected databases and encryption algorithms. Three executions were run: the first execution used a baseline without concurrent workloads, avoiding potential execution time variations measuring the query execution under optimal conditions. This execution is called PowerTest [59]. The following two executions are called Throughput, which evaluates the executions when having concurrent executions checking for time variations [59], evaluating the system's performance as it manages simultaneous requests simulating a more realistic workload, providing insight into the system's scalability and if can maintain the same performance without any high variations in terms of execution times or terms of use of system resources. The Powertest is the cold start, which has no data plans or execution plans in memory, so it doesn't have any record of any past execution to make sure the execution times are unaffected by cache reflecting the raw I/O and optimization costs, without any variation. The Throughput is defined by two executions with different query orders, which inherit the warm plan cache from the Powertest. By changing the order of the executions, it avoided the repeated hot-spot contention reducing time variations, reflecting realistic, concurrent workloads minimizing the cache inducing noise, taking advantage of I/O and CPU parallelism [61].

4.1.1 An overview of TPC-H benchmark

The TPC-H benchmark is a performance benchmark created to evaluate the execution of complex decision-support queries on large-scale databases. It was developed by the Transaction Processing Performance Council (TPC) to simulate real-world business operations in different industries such as retail or manufacturing. The TPC-H benchmark focuses on ad-hoc queries and the ability to evaluate query performance and database systems. Using this benchmark requires the generation of the

data schema and the population of the corresponding data for the chosen scale factor, making it possible to run the 22 business-oriented queries representing typical realistic decision scenarios that involves complex SQL features such as nested subqueries, joins, and aggregations, making it ideal for evaluating system performance under realistic workloads which can be shared in scientific papers and the scientific community [59].

The TPC-H database contains eight tables which include Region, Nation, Supplier, Customer, Orders, Lineitem, Part and Partsupp (as illustrated in Figure 4). The database design includes multiple complex joins and realistic business relationships to support large-scale data volumes. The dss.ri file inside the dbgen folder contains complete schema details along with primary and foreign key information.

TPC-H uses a tool called DBGen which creates different datasets for the different scale factors chosen, the scale factor is what determines the size of the generated dataset, for example a scale factor 0.1 it generates about 100 MB of raw data, for scale factor 1 it generates about 1 GB of raw data, for scale factor 10 it generates about 10 GB, and the same for the upper scale factors and then the generated data is loaded to the TPC-H database schema. TPC-H also provides a tool called QGen to generate variants with randomized substitute parameters, simulating real-world scenarios with new parameters which can affect any query.

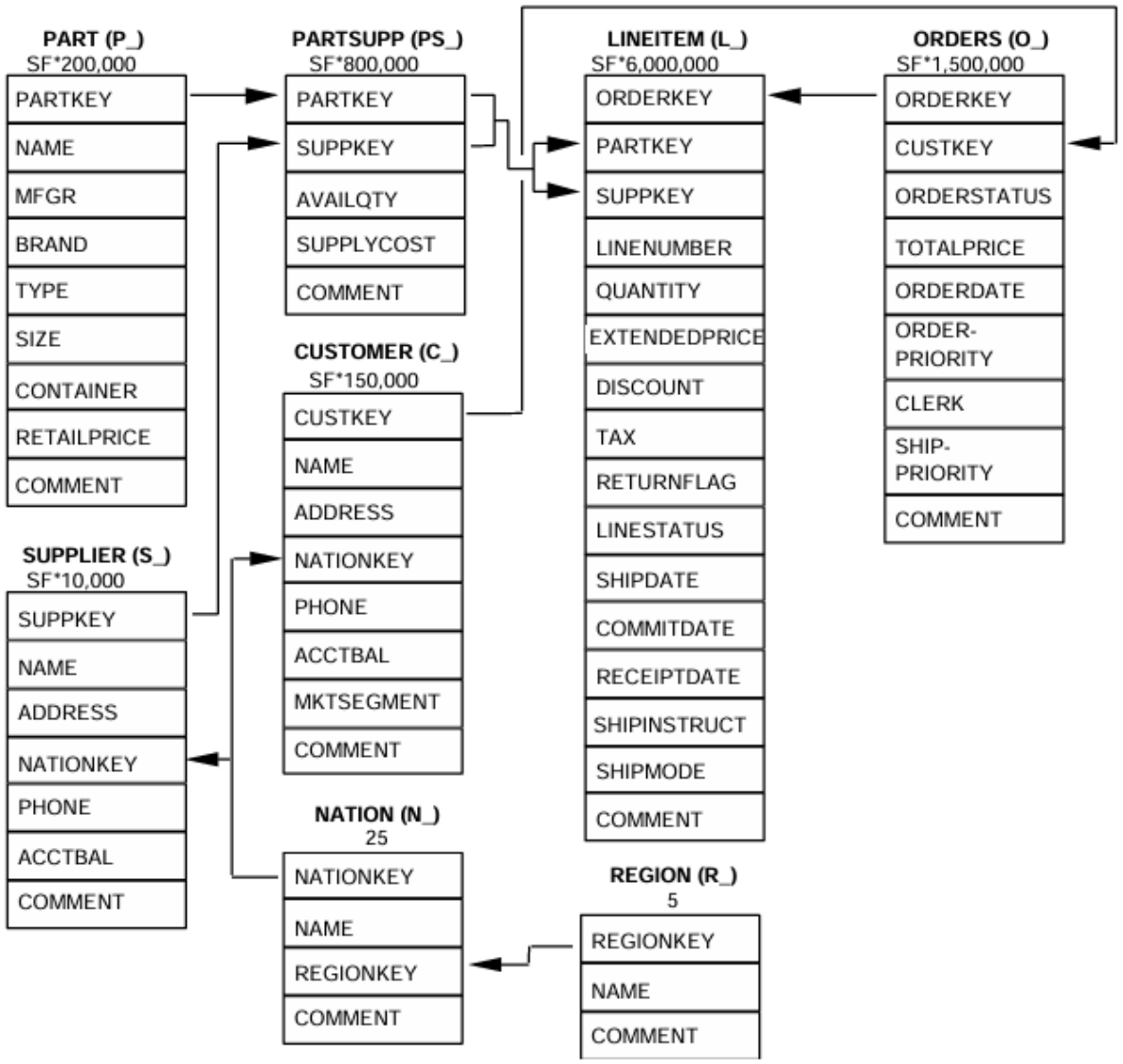


Figure 10 - TPC-H schema model. Retrieved from [59]

4.2 Databases used

To make it possible to test and have results to compare, this study chose some RDBMS, which were Oracle, MySQL and SQL Server, due to the support to TDE and same encryption algorithms [31], [34] and for being recognized databases across different industries and academic research [62], [63] making them strong choices for this evaluation. This work evaluates the encryption performance of the top three databases, as rated by the DB-Engines 2025 ranking, which provides a rank of databases according to their popularity [7], [8], [9]. Oracle, MySQL, and SQL Server consistently appear in the top positions reflecting robustness, support and wide adoption among enterprise organizations and academic environments.

Oracle Corporation created the Oracle database, which is the leading relational database management system based on the DB-Engines 2025 ranking [64]. The

database system supports SQL and PL/SQL programming languages and provides strong reliability features and extensive scalability options [64]. The system provides robustness and scalability features with support and advanced features. The database system provides Real Application Clusters (RAC) and automatic storage management and advanced security options. Oracle introduces new features regularly through its development process and has added machine learning integration and autonomous database functionality and improved cloud environment support in recent times [49].

Oracle provides multiple editions which differ through their licensing requirements. The Oracle Database Express is free while the Oracle Personal Edition requires a license, and the Enterprise Edition and Standard Edition 2 requires a license for its use [64]. Express edition supports only one CPU, 1 GB of RAM, and up to 11 GB of user data, without access to advanced features or security options. Standard Edition 2 is a licensed edition for small to medium applications, limited to 2 CPU sockets and 16 threads, and excludes most enterprise features like TDE or Oracle Data Guard. Enterprise Edition is the most complete and scalable edition, supporting all advanced database options, security features (e.g., TDE, Data Redaction, Database Vault), Real Application Clusters, and Oracle Management Packs. Personal Edition is a single-user licensed edition intended for development use, with the same features as Enterprise Editions except RAC and management packs and is available only for Windows and Linux. In this project it was used Oracle Database 19c Enterprise Edition Version 19.3.0.0.0.

MySQL is an RDBMS that allows storing, managing and accessing large volumes of data efficiently. It was developed originally by MySQL in 1995 and is propriety of Oracle Corporation. MySQL uses SQL, which is the common language to interact with relational databases [49]. In this project, MySQL version 8.0.38 Community Server and MySQL version 9.2.0 Enterprise Server were used. With MySQL v8.0.38 variations were observed along the execution tests, MySQL v9.2.0 was used to see if the same variations would occur or if better results could be achieved compared to earlier versions.

MySQL has a free version called Community Edition, it has Enterprise edition with additional functionalities and support, but it requires a commercial license, although it is free to students to use. Additionally, MySQL offers Cluster Carrier Grade Edition (CGE), a highly-availability version designed for telecom and real-time applications, which is also commercially licensed. MySQL can be used in different operating systems, such as Linux, Windows and macOS, making it flexible and easily implemented. MySQL also has a simple and intuitive interface that makes it easy for beginners and administrators of databases for experienced people. MySQL offers different motors to store, such as InnoDB and MyISAM, which can be selected depending on the project's needs. InnoDB is transactional and supports foreign keys, but MyISAM is faster but does not support transactions. MySQL offers security functionalities, user authentication, and data encryption to protect the information. It concluded that MySQL could be trusted, especially in web applications, content

management and websites that deal with a significant volume of data, making sure that it offers scalability, whether to a small website or large-scale e-commerce, to continue to deliver simplicity and a high performance read and write accesses to the database [49], [65], [66].

Microsoft Corporation developed SQL Server, which is an RDBMS. SQL Server is a product that can be run on Windows and Linux operating systems, and you can also integrate it with Azure Services. It supports SQL and other advanced data security features, such as encryption, data masking and row-level security. It can optimize performance with in-memory processing and a built-in query optimizer that will create different execution plans or query plans. The query optimizer then selects a query plan using a set of heuristics to find a good query plan. SQL Server is used for academic purposes or enterprise applications with enterprise, Standard, Web, Developer, and Express versions [67]. In this project SQL Server version 2022 Developer Edition was used.

4.3 Performance Monitor

Performance metrics were collected using Performance Monitor, used to track and analyze real-time statistics. In this tool were used specifically the "% Processor Time" counter, which tracks the percentage of time the CPU spends executing active (non-idle) threads. Memory usage was analyzed using the "Available Mbytes" counter, reflecting the amount of free physical memory which helped to identify memory pressure during query execution. Disk activity was recorded using the "% Disk Time" counter, indicating how busy the disk was during the query execution.

4.4 Why AES 128, 192, and 256?

AES was chosen as the encryption algorithm to be used because of its widespread adoption, robust security features and availability across different RDMS, including SQL Server and MySQL [31], [34], [37]. AES is also known as the global encryption standard by NIST [68]. Its adoption by many industries means that it has a high level of trust and reliability. One of the other reasons is that AES is natively supported by different RDBMS platforms, including Oracle, MySQL and SQL Server, making it a good choice for comparative analysis of encryption performance and implementation. The only downside is that MySQL does not support AES-128 and AES-192 natively. In MySQL defaults to AES-256 and does not refer any support to AES-128 and AES-192 when applying transparent data encryption [69].

Using AES, an environment is created where it is possible to compare performance and behavior across the selected platforms, ensuring consistency when evaluating.

4.5 Experimental Environment

To do the experiments without interference from other processes, virtual machines were requested to ISEC, which provided three virtual machines for this purpose. Each virtual machine had its operating system, Windows 10 Pro, which had 16GB of RAM, an Intel(R) Xeon(R) CPU E5-2660 v2 @ 2.20GHz, and at least 100GB of storage. For this study, Microsoft SQL Server 2022 (version 16.0.1130.5 (X64) – RTM-GDR (KB5046057)) was installed in Developer Edition on one of the virtual machines. For MySQL was used the versions 8.0.38 and 9.2.0 enterprise version and installed in another separated virtual machine. To evaluate Oracle, was used the version 19c Enterprise Edition Version 19.3.0.0.0 which was installed in one of the three machines as well. These virtual machine's purpose was to isolate any noise from other processes and get clearer results, making it a dedicated environment for testing. The configuration parameters applied to each DBMS setup, including memory allocation settings (e.g., Oracle's SGA), are detailed in Appendix E.

4.6 Data Upload Process

This experiment used the TPC-H benchmark, a well-known tool, to simulate decision support systems and perform benchmarks for the performance of a database management system. In the TPC-H it was used DBGen tool to create different datasets for the selected scale factors. For each database system, the datasets were uploaded using different processes:

- Oracle: The data was loaded via LOAD DATA INFILE and the command was saved in a ctl file and executed in command line "sqlldr userid=system/"db password"@//127.0.0.1:1521/orcl control="pathfile until the folder with the insert commands"\region.ctl.txt log=load.log"
- MySQL: Data is loaded via LOAD DATA INFILE and optimized to insert lots of data.
- SQL Server: The data were loaded via BULK INSERT and the typical optimizations to make those data as efficient as possible.

Since the environment was shared, some of the processes may have variability in execution time with respect to their parallel running workloads. Several test runs were conducted to cancel out this effect and ensure the consistency of results.

4.7 Database Size Variation with and without Encryption

This section analyses the differences among different database platforms per scale factor and encryption algorithm.

Encryption often increases the overall database size when encrypted. However, transparent data encryption is not the case in the Oracle [70], MySQL [37], and SQL Server [31] as shown in Table 3. This result may be attributed to how database handles encryption at storage level and the TDE itself. TDE encrypts data files transparently without adding extra storage overhead. Although encryption is expected to increase the size of the database, these platforms maintain the same size due to transparent encryption.

Table 3 – Encrypted database and No Encrypted Database Size

Database Platform	Scale Factor	Encryption Algorithm	Database size (MB)
Oracle	0.1	Non-encryption	146.31
Oracle	0.1	AES-128	146.31
Oracle	0.1	AES-192	146.31
Oracle	0.1	AES-256	146.31
Oracle	1	Non-encryption	1412.5
Oracle	1	AES-128	1412.5
Oracle	1	AES-192	1412.5
Oracle	1	AES-256	1412.5
Oracle	10	Non-encryption	14175.25
Oracle	10	AES-128	14175.25
Oracle	10	AES-192	14,175.25
Oracle	10	AES-256	14,175.25
MySQL	0.1	Non-encryption	154.9
MySQL	0.1	AES-256	154.9
MySQL	1	Non-encryption	1,531.9
MySQL	1	AES-256	1,531.9
SQL Server	0.1	Non-encryption	272
SQL Server	0.1	AES-128	272
SQL Server	0.1	AES-192	272
SQL Server	0.1	AES-256	272
SQL Server	1	Non-encryption	3,024
SQL Server	1	AES-128	3,024
SQL Server	1	AES-192	3,024
SQL Server	1	AES-256	3,024
SQL Server	10	Non-encryption	20,176
SQL Server	10	AES-128	20,176
SQL Server	10	AES-192	20,176
SQL Server	10	AES-256	20,176

4.8 Encryption Process

In this section, we present the encryption processes of Oracle, MySQL and SQL Server that secure the data. Both databases support TDE, which was used in the experiment to avoid changing the application layer by ensuring the data is encrypted at rest.

4.8.1 Encryption in MySQL using Tablespace Encryption

MySQL supports TDE at the tablespace level, which is available with MySQL Enterprise Edition and a Community Edition. This experiment used tablespace encryption to encrypt data in MySQL's InnoDB tables. To enable encryption, the following steps have been followed:

For MySQL Community Edition, first, it was necessary to enable the keyring plugin, in this case the plugin keyring_file, which stores the encryption keys outside of the database tables, "early-plugin-load=keyring_file.dll keyring_file_data='place where keyring is installed"', this commands are being executed in the my.cnf file, during server startup, which is in the installation folder and is executed to make the plugin available for the user to apply encryption to a tablespace.

Second when wanting to encrypt a tablespace, this is the command to apply: "CREATE TABLESPACE tpch_tde ADD DATAFILE 'tpch_tde.ibd' ENCRYPTION='Y';". When creating a table under this tablespace, the table will automatically inherit the encryption settings from the tablespace. However, it is necessary to mention the tablespace name during table creation, where it is needed to leave this command "TABLESPACE `tpch_tde`".

For MySQL Enterprise Edition, the way it was made the encryption enabled in the database system is different from the MySQL Community Edition, where it is not done on the my.cnf file, but it is done by creating a manifest to tell which keyring component to load, this file will be in the data folder under the MySQL folder which is in the ProgramData directory:"{"components":"C:\Program Files\MySQL\mysql-commercial-9.2.0-winx64\lib\plugin\component_keyring_file.dll"}". After that, two other files are created, one in the plugin folder which is responsible for giving the path to where the component keyring will be placed which is called the local file and has this content: "{"path": "/usr/local/mysql/keyring/component_keyring_file", "read_only": false}". The other file is the global file which says to the system to read local manifest: "{"read_local_manifest": true}".

In the shared environment, encryption may burden the system more under high I/O load conditions. Therefore, benchmarks have been repeated to measure performance degradation due to encryption under various system conditions.

4.8.2 Encryption in SQL Server using TDE

In SQL Server, TDE performed encryption on a whole database level. TDE allows for the encryption of not only data files but also log files. This way, all information that rests on the disk will remain encrypted. Encryption in SQL Server takes place in the following steps:

In this experiment, TDE was applied to SQL Server to encrypt the whole database. This means the data at rest would be kept safe. The establishment and configuration of TDE involve a few essential steps that establish and secure a hierarchy of encryption mechanisms. This makes the data in the database and transaction logs encrypted, and none of this sensitive information may come into any unauthorized sources' access, even in those cases where the physical files are compromised.

First of all, encryption starts with Master Key creation. This master key becomes pivotal in securing the encryption infrastructure of SQL Server. The master key is stored in the master database and encrypted by a password. It plays a highly protective role in safeguarding the other components of encryption that come afterwards. Without this master key, the initialization of the database encryption mechanism cannot be done.

Once the master key is created, the following procedure will be the creation of a certificate. The certificate also resides within the master database. It encrypts the DEK, which will encrypt the database's actual data. The certificate protects the DEK so that even though the database files are accessed, the data therein remains unreadable without the certificate.

Once the certificate is obtained, a Database Encryption Key (DEK) is generated within the database, which encrypts the actual database, including the main data files and the transaction logs. The DEK uses the AES encryption algorithm, one of today's most robust and used standards, since it presents the best balance between security and performance. Once the DEK is generated, it gets wrapped by the previously created certificate, adding an extra level of protection.

Finally, encryption is activated at the database level as TDE with the DEK in place. TDE is activated by informing the SQL Server to SET ENCRYPTION ON for the database.

From now on, any data written to disk in the main database file or the transaction logs are encrypted using the DEK. Encryption is transparent, requiring no changes to the application or user queries. Data are automatically encrypted as they are written to disk and decrypted as they are read into memory.

Along with encrypting the database, TDE also encrypts the log files. Any backups created from this encrypted database will also be automatically encrypted. This becomes an important security measure for those scenarios where the backups may need to be stored off-site or in environments where unauthorized access could be an issue. If compromised, these encrypted backups ensure the data remains protected. Perform steps in reverse when turning encryption off may be necessary,

including removal of the associated keys and certificates. In this case, encryption needs to be turned off first, which will already accomplish turning TDE off and stop automatic data encryption. Once encryption is turned off, the DEK can safely be dropped from the database. Once the DEK is removed, the self-signed certificate used to protect the DEK is no longer needed and can be dropped from the master database. Finally, once the certificate has been dropped, the master key can be removed, entirely removing the encryption infrastructure.

4.8.3 Encryption in Oracle using TDE

Transparent Data Encryption is a feature used to encrypt data at rest in order to protect sensitive information in tablespaces and columns. This way encryption ensures that even if database files are accessed outside of the Oracle environment.

To encrypt a tablespace, it was necessary first to follow some steps to make it possible to have encryption ready to use in the database engine.

Oracle asks for a wallet, a secure location that stores the TDE master key. Before that it was needed to set some environment variables, keystore initializations, creation and once that is acquired it is possible to create encrypted tablespace, and the rotation of key sizes as needed.

First it is important to tell Oracle where the database binaries live, so it was set the variable in command line ORACLE_HOME where my installation file lived set ORACLE_HOME=C:\oracle19c. Normally Oracle uses a binary "spfile" to read the settings at startup, so it was create a pfile "CREATEPFILE='…/admin/orcl/pfile/orcl-2023-08-14.ora'", this file has all parameters with the settings. This step is not required for TDE, but it's good DBA practice. After this we are ready to create the wallet, a secure container for the master encryption key "ADMINISTER KEY MANAGEMENT CREATE LOCAL AUTO_LOGIN KEYSTORE FROM KEYSTORE 'C:\oracle19c\admin\orcl\WALLET\TDE' IDENTIFIED BY "your_wallet_password";" and now it's ready to create encrypted tablespaces "CREATE TABLESPACE tde_secure_tbsp DATAFILE '…\oradata\ORCL\tde_secure1.dbf' SIZE 15G ENCRYPTION USING 'encryption_algorithm' DEFAULT STORAGE(ENCRYPT);" if we want to change the encryption key of the tablespace is necessary to rotate the encryption key on the tablesapce, usint the command alter, ALTER TABLESPACE tde_secure_tbsp ENCRYPTION ONLINE USING 'encryption_algorithm' REKEY;

4.9 Automated Data Extraction and Multi-Query Execution Management with Python

This sub-chapter explains the Python script created to generate a script to be executed in each of the database platforms and the data extraction script to cycle

through the execution times extracted from the respective database platforms console output and converting them into an excel file (CSV).

The TPC-H already has a list of execution queries arranged in specific orders for testing purposes. To maintain consistency, these functions are already prepared to rearrange the order to match the values to the respective query number. However, the queries must be executed in the exact order specified by the TPC-H benchmark.

4.9.1 Multi-Query Execution Management with Python

To execute multiple-query executions, it was necessary to build functions which could return a script that could execute the queries in one execution called Powertest and run multiple executions called Throughput. We built scripts for MySQL, SQL Server, and Oracle because the syntax in either one or another is different, and some changes are necessary to run without errors. The functions for SQL Server were ReturnThroughputTest() and ReturnPowerTest(). The functions for MySQL were ReturnMySQLThroughputTest() and ReturnPowerTestMySql(). The functions for Oracle were ReturnThroughputTestOracle() and ReturnPowerTestOracle(). These functions are provided in Appendix A for reference. In Figure 11 and 12 is placed a generalized fluxogram that shows how the code developed works, respectively to build the Powertest script with the correct order of queries and to build the Throughput script with the correct order by execution array number and by query number.

ReturnThroughputTest() prints at the beginning "SET STATISTICS TIME ON;" to activate the print to the console the elapsed time of each query and to disable it is printed at the end of the script "SET STATISTICS TIME OFF;". To make it possible when doing data extraction to see when an execution of 22 queries starts, it is printed "ThroughputTest-X-begin", and when an execution finishes "ThroughputTest-X-end". This is for each execution cycle. To identify the extracted query number, it is printed before the execution the query "Query X-Begin" and after the execution of the query "Query X-End" (the X means the query number).

ReturnPowerTest() is the same as ReturnThroughputTest(), where it prints the line of code to enable the print to the console of the elapsed time of each query and at the end of the script, it disables the printing of the elapsed time for each new execution. To start the powertest and to identify when the data extraction of the powertest result begins, it is printed "PowerTest-X-begin". To identify each query to extract the elapsed time, it is printed before the query execution "Query X-Begin" and after the execution "Query X-End". The powertest script finishes with "PowerTest-End", and the statistics are disabled.

ReturnMySQLThroughputTest() does not do prints; it uses comment markers to organize and label the script for readability. This function's purpose is to create a MySQL script containing the specified number of executions, which in each execution is 22 queries by the respective order to the respective execution number.

ReturnPowerTestMySql() is the same as ReturnMySQLThroughputTest(). However, it just appends to the script the first execution with the respective order of the 22 queries with the comment markers to organize and label the script for readability.

ReturnThroughputTestOracle() prints at the beginning the variable setOracleStatisticsOn which contains the code to print the elapsed times of each query, the code to redirect the execution times to a text file, and the date time of when the script execution started. To disable and to close the printing to a text file it is printed the setOracleStatisticsOff variable. This function also appends the queries to the specified number of executions, which in each execution is 22 queries by the respective order to the respective execution number.

ReturnPowerTestOracle() is the same as ReturnThroughputTestOracle (). However, it just appends to the script the first execution with the respective order of the 22 queries with the comment markers to organize and label the script for readability.

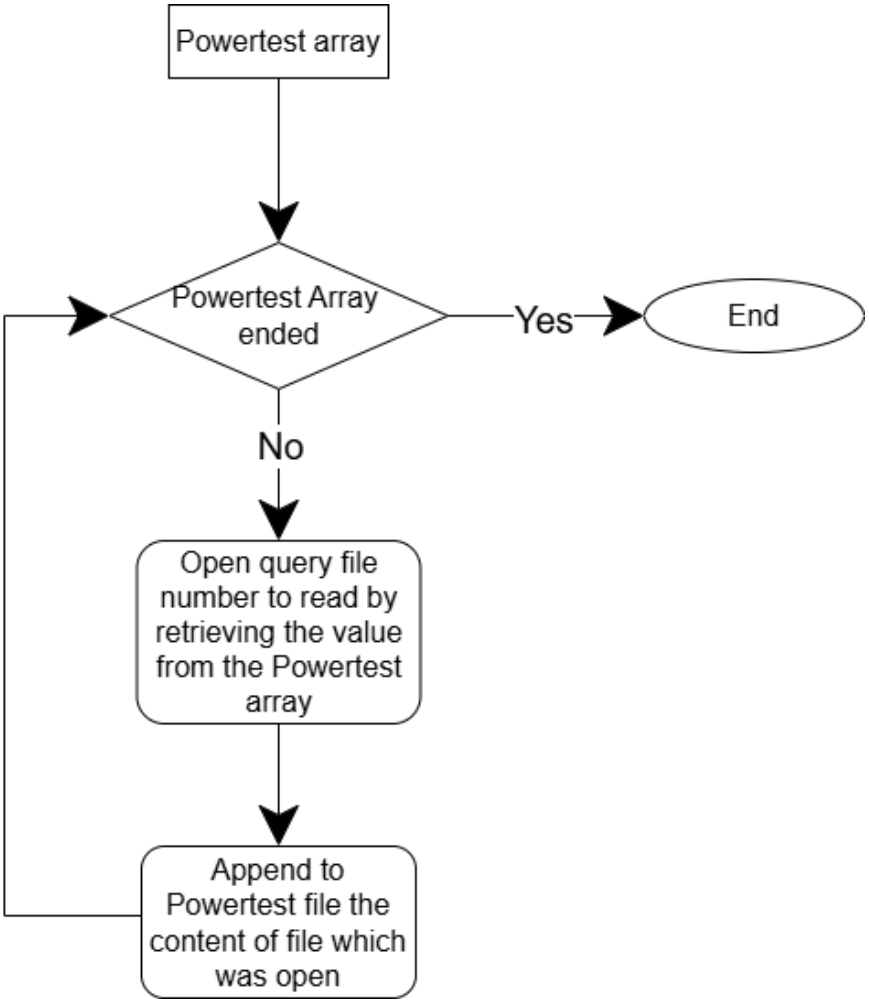


Figure 11 - Generalized fluxogram of Powertest representing the functional flow of the developed code.

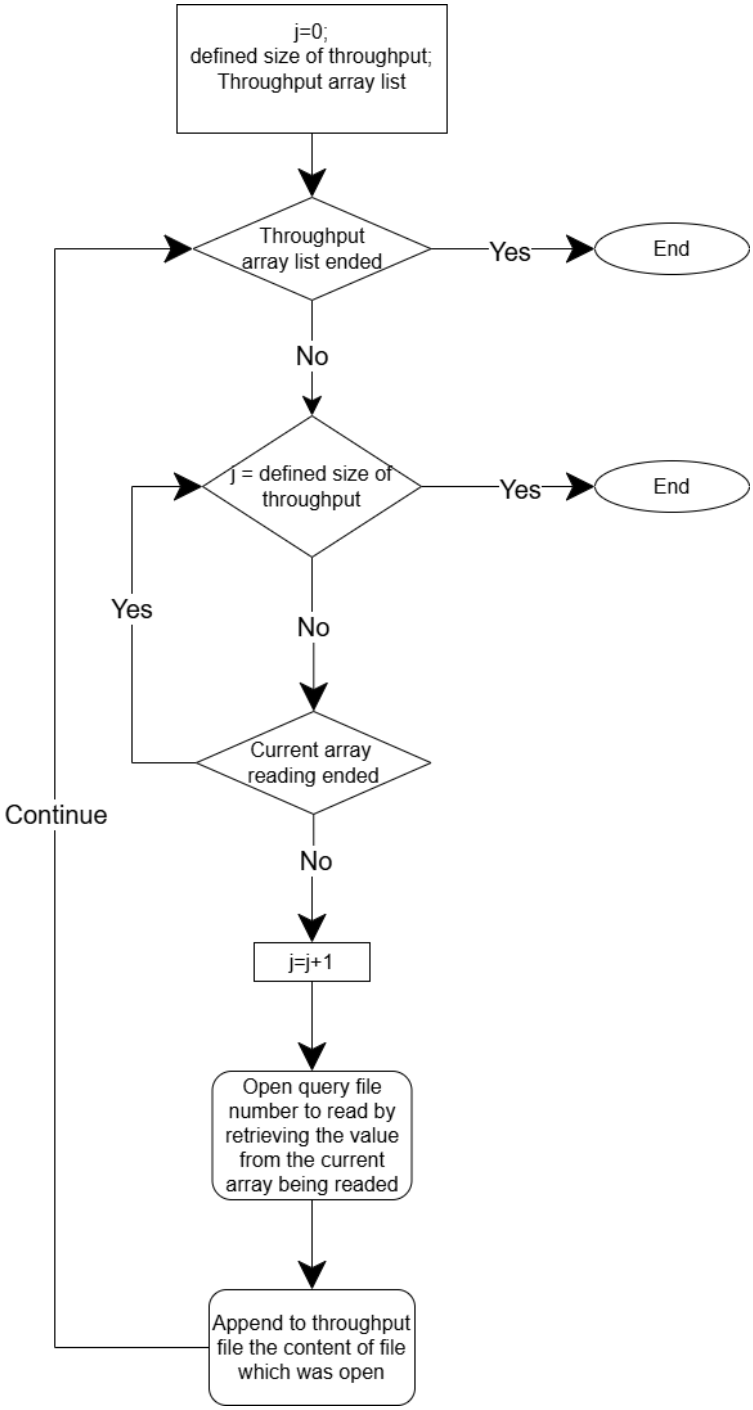


Figure 12 – Generalized fluxogram of Throughput representing the functional flow of the developed code.

4.9.2 Automated Data Extraction with Python

To extract the execution data times from the text file (first, there was the need to copy the output from the console to a text file), a script was developed to handle Powertest and Throughput files. The function to extract the powertest and throughput from a SQL Server were ConvertValuesToExcelPowertest() and ConvertValuesToExcel(), respectively and the function to extract MySQL powertest

throughput were ConvertValuesMySQLToExcelPowertestTeeVersion() and ConvertValuesMySQLToExcelThroughputTeeVersion () respectively, and the function to extract Oracle powertest and throughput were ConvertValuesOracleToExcelPowertest() and ConvertValuesOracleToExcelThroughput(). These functions are provided in Appendix B for reference. In Figure 13 and 14 is placed a generalized fluxogram that shows how the code developed works, respectively to convert the Powertest txt file to excel with the elapsed times in the correct order and to convert the Throughput txt file to excel with the elapsed times with the correct order by execution array number and by query number, as designed by TPC-H [59].

ConvertValuesToExcel() processes Throughput Test results from the SQL Server console. It reads line by line, looking for "throughputtest" which indicates a new execution is starting, extracting the elapsed time for each query from Query1 to Query22. After finishing each execution, there is a calculation for the total and average of the respective execution. At the end of having all the arrays with each execution, it is written in the CSV file for each array.

ConvertValuesToExcelPowertest() processes the Powertest results from the SQL Server console by reading line per line as the function to extract the Throughput Test; the difference is that it will read only one execution of 22 queries. When reading line per line, it looks for "powertest" and "begin" to know when the execution starts and extracts the elapsed times until it finds "end", which means that the execution is finished.

ConvertValuesMySQLToExcelPowertestTeeVersion() processes the Powertest results from the MySQL console by reading line per line trying to find "row in set" or "rows in set" or "Empty set", which indicates that it is a line to extract elapsed time or in case it is empty state it is because there was no time to be extracted. When the script extracts an execution time from that line, it converts the time from seconds to ms. It is inserted in an array in the specific position of the respective query to have an exact order in Excel by the query number itself. At the end of cycling through the text file, the total and average for all queries are calculated, providing a summary of execution times.

ConvertValuesMySQLToExcelThroughputTeeVersion() processes the throughput test results from the MySQL console by reading line per line having the same formula as ConvertValuesMySQLToExcelPowertestTeeVersion(). However, in this case, it is creating additional arrays that handle multiple query executions.

ConvertValuesOracleToExcelPowertest() processes the Powertest results from Oracle console by reading line per line trying to find the "Elapsed:" extracting times in the format hh:mm:ss.ms. When the script extracts an execution time from that line, it converts the time toms using this formula "elapsed_time_ms = (hours * 3600 + minutes * 60 + seconds) * 1000". It is inserted in an array in the specific position of the respective query to have an exact order in Excel by the query number itself. At the end of cycling through the text file, the total and average for all queries are calculated, providing a summary of execution times.

ConvertValuesOracleToExcelThroughput() processes the throughput test results from the MySQL console by reading line per line having the same formula as ConvertValuesOracleToExcelPowertest (). However, in this case, it is creating additional arrays that handle multiple query executions.

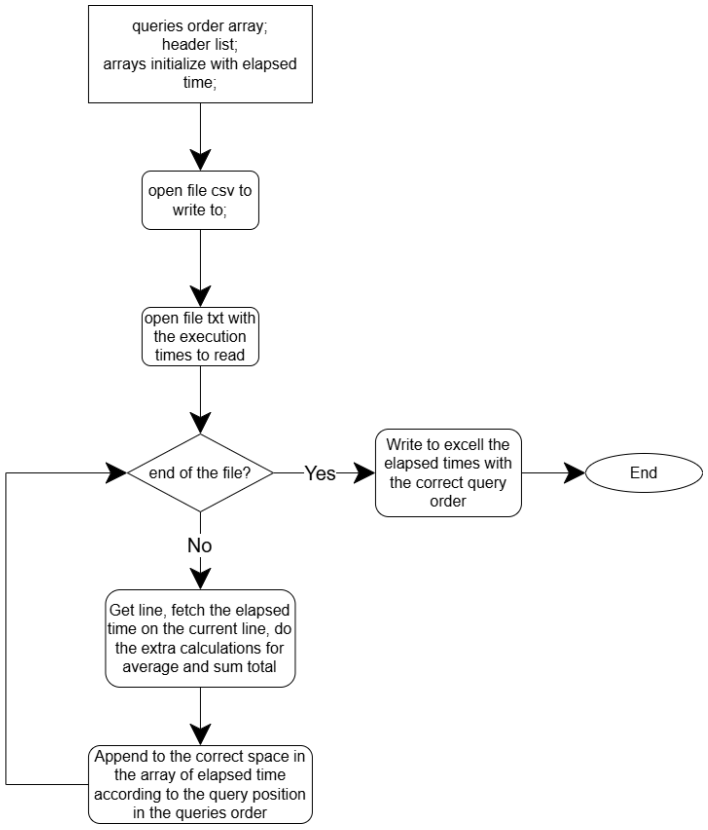


Figure 13 – Workflow for converting Powertest execution times from a TXT file to Excel

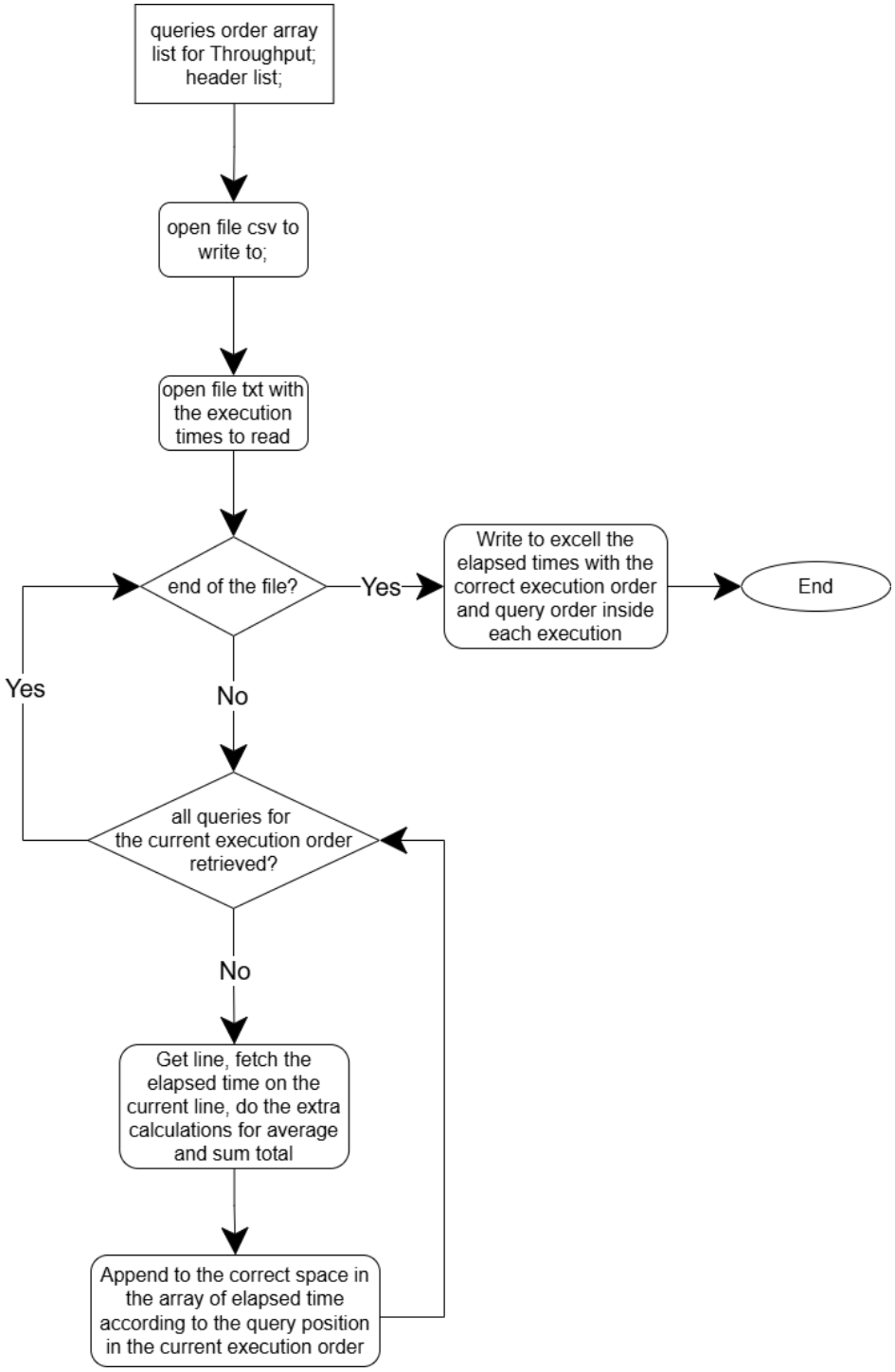


Figure 14 - Workflow for converting Throughput execution times from a TXT file to Excel

4.9.3 Auxiliary Functions in Python

This section explains essential parameter definitions used in the main scripts, which are appended to Appendix A and Appendix B. These commands are listed in Appendix C.

For query execution order, two lists were created named "powertest" and "ThroughputTest", which specify the order in which the queries should be executed, created

using the TPC-H guideline. The number zero in the first position of each list indicates a file containing initial setup commands, such as views or initializing other required code to be executed at the beginning of the script created. The throughput script will only print the file information one time at the beginning of the script.

To define how many executions are necessary for the script file with multiple executions, the "sizeOfThroughput" variable is used. This allows for controlled batch sizes. To define where to read and where to write, the "fromFolder" and "toFolder" variables are used, respectively.

The variables "pathEncrypted", "filenameTh", "filenameCSVTh", "filenamePt", and "filenameCSVPt" define the paths and file names used when converting the extract execution times from the consoles into Excel CSV files.

Commands to be printed in the scripts were also created. The variable "timeToShow" is used to print the time the test started and print at the end of the script to know when it is finished. The variable "setStatisticsOn" and the variable "setStatisticsOff" are commands to enable and disable SQL Server's "SET STATISTICS TIME" option, which records query execution time. The variable "setOracleStatisticsOn" and the variable "setOracleStatisticsOff" are variables with commands to enable and disable Oracle printing of elapsed times to a text file.

5 EXPERIMENTAL EVALUATION

This section presents an analysis of the performance of various encryption algorithms across different database platforms, evaluated at specific scale factors. The primary objective is to identify the most efficient encryption approach for different scenarios, considering the computational overhead introduced by the encryption algorithms and the scalability of each platform. Each platform implements standard encryption methods, enabling a consistent and comparable assessment. Performance is evaluated by executing 22 TPC-H benchmark queries three times for each scale factor and encryption algorithm, with key metrics such as query execution time, CPU usage, memory consumption, and disk time percentage being analyzed.

The execution sequence is different for each test run. The Powertest has the following order of the 22 queries:

14, 2, 9, 20, 6, 17, 18, 8, 21, 13, 3, 22, 16, 4, 11, 15, 1, 10, 19, 5, 7, 12.

The Throughput Test executes its first sequence as follows:

21, 3, 18, 5, 11, 7, 6, 20, 17, 12, 16, 15, 13, 10, 2, 8, 14, 19, 9, 22, 1, 4.

The second execution sequence differs from the first one as follows:

6, 17, 14, 16, 19, 10, 9, 2, 15, 8, 5, 22, 12, 7, 13, 18, 1, 4, 20, 3, 11, 21.

These metrics provide valuable insights into how databases handle encryption under varying conditions and establish a direct correlation between the database management system's capabilities and the overall efficiency of the encryption processes.

5.1 Total Execution sum

Table 4 summarizes the experiments and metrics used to compare the performance of the queries, including the total execution time of the 22 queries, and resource usage. The scale factor represents the size of the dataset used in the benchmark, with "S0.1" corresponding to 100 MB and "S1" to 1 GB. The database platform refers to the environment where the queries were executed. At the same time, the test number indicates the iteration of the query executions. The encryption algorithm column specifies which algorithm was applied in each case.

Execution time is measured as the cumulative time, in ms, to run all 22 queries. CPU usage is monitored using the Windows Performance Monitor, specifically the "% Processor Time" counter, which tracks the proportion of time the CPU spends executing non-idle threads. Memory usage is observed using the "Available Mbytes" counter, reflecting the amount of physical memory available for system use, helping to determine if additional memory might be required. Disk usage is recorded using the "Disk Time Percentage" counter, indicating how busy the disk is during the query executions.

This detailed evaluation offers a comprehensive understanding of the relationship between encryption performance and the underlying resource usage of the database management system.

The following is a breakdown of the results based on different factors and encryption configurations.

Table 4 – Performance Results of Database Platforms Using AES Encryption

SF	Database	# Test	Encryption algorithm	Execution (ms)	Stdev (ms)	RAM (%)	CPU (%)	Disk Time (%)
0.1	Oracle	1	no-encryption	11,860	480	34.65	50.84	1.14
0.1	Oracle	2	no-encryption	6,970	184	55.09	26.70	0.39
0.1	Oracle	3	no-encryption	6,270	167	55.09	26.70	0.39
0.1	Oracle	1	AES-128	15,230	755	55.02	35.27	0.47
0.1	Oracle	2	AES-128	6,330	172	27.66	41.11	0.86
0.1	Oracle	3	AES-128	6,050	155	27.66	41.11	0.86
0.1	Oracle	1	AES-192	15,380	687	26.15	40.63	23.95
0.1	Oracle	2	AES-192	7,100	188	27.80	57.17	29.89
0.1	Oracle	3	AES-192	4,230	180	27.80	57.17	29.89
0.1	Oracle	1	AES-256	14,130	603	28.24	6.85	3.43
0.1	Oracle	2	AES-256	6,560	175	28.26	28.17	1.28
0.1	Oracle	3	AES-256	6,360	161	28.26	28.17	1.28
1	Oracle	1	no-encryption	293,030	23,191	35.71	32.04	216.73
1	Oracle	2	no-encryption	170,450	13,415	51.92	31.05	1.42
1	Oracle	3	no-encryption	169,410	13,297	51.92	31.05	1.42
1	Oracle	1	AES-128	1,000,840	139,914	49.91	22.61	62.72
1	Oracle	2	AES-128	986,750	137,678	54.64	37.01	60.65
1	Oracle	3	AES-128	994,490	139,166	54.64	37.01	60.65
1	Oracle	1	AES-192	990,710	24,726	45.48	18.97	111.03
1	Oracle	2	AES-192	942,180	23,620	52.56	19.10	135.80
1	Oracle	3	AES-192	894,150	25,925	52.56	19.10	135.80
1	Oracle	1	AES-256	1,300,740	211,410	42.02	27.97	92.09
1	Oracle	2	AES-256	482,490	67,590	47.38	27.18	0.49
1	Oracle	3	AES-256	480,970	66,458	47.38	27.18	0.49
10	Oracle	1	no-encryption	1,446,090	38,512	47.61	11.75	53.09
10	Oracle	2	no-encryption	1,287,610	34,109	50.05	15.89	66.16
10	Oracle	3	no-encryption	1,335,240	37,029	50.05	15.89	66.16
10	Oracle	1	AES-128	1,532,510	38,895	57.08	38.81	58.77
10	Oracle	2	AES-128	1,555,030	41,857	54.64	37.01	60.65
10	Oracle	3	AES-128	1,534,490	43,417	54.64	37.01	60.65
10	Oracle	1	AES-192	2,076,110	80,479	32.37	11.93	48.62
10	Oracle	2	AES-192	1,812,170	67,663	53.75	29.22	70.75
10	Oracle	3	AES-192	1,742,090	72,918	53.75	29.22	70.75
10	Oracle	1	AES-256	10,614,900	685,833	58.63	29.21	61.73

SF	Database	# Test	Encryption algorithm	Execu-tion (ms)	Stdev (ms)	RAM (%)	CPU (%)	Disk Time (%)
10	Oracle	2	AES-256	8,951,690	519,565	60.71	29.89	32.73
10	Oracle	3	AES-256	4,807,050	397,167	60.71	29.89	32.73
0.1	MySQL CE v8.0.38	1	no-encryption	97,690	5,926	33.23	29.32	79.86
0.1	MySQL CE v8.0.38	2	no-encryption	63,730	2,840	33.48	36.10	31.95
0.1	MySQL CE v8.0.38	3	no-encryption	55,930	3,415	33.48	36.10	31.95
0.1	MySQL CE v8.0.38	1	AES-256	37,400	1,443	32.77	40.45	7.93
0.1	MySQL CE v8.0.38	2	AES-256	33,900	1,447	32.77	42.47	7.45
0.1	MySQL CE v8.0.38	3	AES-256	33,310	1,389	32.77	42.47	7.45
1	MySQL CE v8.0.38	1	no-encryption	1,997,740	155,857	33.08	25.29	59.71
1	MySQL CE v8.0.38	2	no-encryption	1,917,410	140,040	33.28	17.27	59.48
1	MySQL CE v8.0.38	3	no-encryption	1,936,610	140,711	33.28	17.27	59.48
1	MySQL CE v8.0.38	1	AES-256	1,985,820	150,195	35.26	28.60	95.53
1	MySQL CE v8.0.38	2	AES-256	2,135,410	149,519	35.33	18.17	36.97
1	MySQL CE v8.0.38	3	AES-256	1,977,470	145,986	35.33	18.17	36.97
0.1	MySQL EE V9.2	1	no-encryption	9,620	424	44.08	49.17	45.14
0.1	MySQL EE V9.2	2	no-encryption	9,770	404	44.21	50.14	14.33
0.1	MySQL EE V9.2	3	no-encryption	10,350	424	44.21	50.14	14.33
0.1	MySQL EE V9.2	1	AES-256	12,590	669	31.76	25.22	70.17
0.1	MySQL EE V9.2	2	AES-256	10,690	451	32.51	20.20	72.16
0.1	MySQL EE V9.2	3	AES-256	9,190	386	32.51	20.20	72.16
1	MySQL EE V9.2	1	no-encryption	1,094,450	79,307	36.64	17.66	54.97
1	MySQL EE V9.2	2	no-encryption	821,380	60,573	36.71	16.36	68.37
1	MySQL EE V9.2	3	no-encryption	788,530	52,522	36.71	16.36	68.37
1	MySQL EE V9.2	1	AES-256	1,103,500	68,805	31.76	25.22	70.17

SF	Database	# Test	Encryption algorithm	Execu-tion (ms)	Stdev (ms)	RAM (%)	CPU (%)	Disk Time (%)
1	MySQL EE V9.2	2	AES-256	1,088,000	84,303	43.64	12.92	71.97
1	MySQL EE V9.2	3	AES-256	1,093,440	82,096	43.64	12.92	71.97
0.1	SQL Server	1	no-encryption	8,245	535	55.24	36.87	0.78
0.1	SQL Server	2	no-encryption	7,443	500	55.09	26.70	0.36
0.1	SQL Server	3	no-encryption	7,605	504	55.09	26.70	0.36
0.1	SQL Server	1	AES-128	7,466	529	55.02	35.27	0.47
0.1	SQL Server	2	AES-128	6,974	527	55.04	44.19	0.63
0.1	SQL Server	3	AES-128	7,249	514	55.04	44.19	0.63
0.1	SQL Server	1	AES-192	6,031	339	37.68	45.57	0.87
0.1	SQL Server	2	AES-192	6,346	478	37.85	62.96	0.98
0.1	SQL Server	3	AES-192	3,849	239	37.85	62.96	0.98
0.1	SQL Server	1	AES-256	5,402	293	36.86	53.43	2.45
0.1	SQL Server	2	AES-256	6,825	429	37.15	42.54	0.88
0.1	SQL Server	3	AES-256	7,030	485	37.15	42.54	0.88
1	SQL Server	1	no-encryption	19,912	1,717	50.39	38.43	0.34
1	SQL Server	2	no-encryption	18,505	1,452	50.52	39.35	0.32
1	SQL Server	3	no-encryption	18,687	1,625	50.52	39.35	0.32
1	SQL Server	1	AES-128	20,391	1,765	54.98	50.98	1.07
1	SQL Server	2	AES-128	18,752	1,448	54.99	56.73	0.65
1	SQL Server	3	AES-128	15,311	1,083	54.99	56.73	0.65
1	SQL Server	1	AES-192	19,921	1,822	54.73	49.34	0.40
1	SQL Server	2	AES-192	20,369	1,527	54.73	56.04	0.77
1	SQL Server	3	AES-192	17,589	1,488	54.73	56.04	0.77
1	SQL Server	1	AES-256	24,304	1,801	40.59	50.27	8.34
1	SQL Server	2	AES-256	18,554	1,387	43.67	32.23	0.49
1	SQL Server	3	AES-256	18,320	1,559	43.67	32.23	0.49
10	SQL Server	1	no-encryption	164,421	6,994	82.32	94.36	19.80
10	SQL Server	2	no-encryption	161,538	7,264	82.73	94.02	19.48
10	SQL Server	3	no-encryption	152,121	6,856	82.73	94.02	19.48
10	SQL Server	1	AES-128	187,209	7,458	93.95	90.36	99.98
10	SQL Server	2	AES-128	174,192	7,372	95.65	87.45	69.55
10	SQL Server	3	AES-128	170,865	7,474	95.65	87.45	69.55
10	SQL Server	1	AES-192	179,812	6,809	95.53	93.36	132.57
10	SQL Server	2	AES-192	171,577	7,271	95.82	93.49	94.96
10	SQL Server	3	AES-192	175,818	7,157	95.82	93.49	94.96
10	SQL Server	1	AES-256	225,773	8,722	91.59	68.43	29.14
10	SQL Server	2	AES-256	198,004	8,794	94.11	77.75	59.98
10	SQL Server	3	AES-256	183,100	8,304	94.11	77.75	59.98

For a scale factor of 0.1 in SQL Server, the total execution time for no-encryption ranges from 7,443ms to 8,245ms. No-encryption is expected to result in faster execution times than the encrypted versions. However, AES-192 and AES-256 sometimes have lower execution times. For example, in one of the tests, AES-192 returned a total execution time of 3,849ms, outperforming the lowest time for non-encryption results, 7,443ms. having a more stable execution time. This can be explained by the scale factor number which can give more variations in terms of execution times and system performance measurements and encryption operations can benefit from better memory/cache consumption, giving faster results. No-encryption is expected to result in faster executions than the encrypted versions. However, encryption algorithms, such as AES-192 and AES-256, have sometimes lower execution times.

AES-192 stands as the fastest average execution time at Scale Factor 0.1 for SQL server, but not so predictable with a standard deviation of 1,360 ms compared to AES-128, AES-256, and no-encryption with 247 ms, 887 ms, and 424 ms respectively. AES-192 might be preferred when minimizing execution time is critical, but it comes with a high variability. AES-192 produces the minimum total execution time of 3,849ms during this test, performing better than all encryption methods and no-encryption method regarding total execution time. AES-192 requires additional CPU resources to operate, but it uses up to 62.96% of CPU power while keeping execution times at reasonable levels under this specific load condition. AES-192 stands as the best encryption solution when organisations require moderate security strength without excessive performance degradation. AES-128 produces longer execution times than AES-192 despite its slightly lower CPU and disk time usage, which makes it a less effective choice. AES-256 operates as a strong encryption method, but its longer execution times make it unsuitable for applications requiring high performance in lighter workloads.

When the scale factor increases to 1 for SQL Server, AES-192 offers slightly more consistent execution with a lower standard deviation compared to AES-128, though not the fastest option. It gives a good balance between stability and security, with total execution times ranging from 17,589ms to 20,369ms and moderate CPU usage compared to AES-256, though AES-128 has faster average times. AES-256 is, by default, the most secure algorithm due to its key size and number of encryption rounds, which, as expected, produces the longest total execution time of 24,304 ms and the highest disk time measurement, reaching 8.34%, which reduces its practicality in performance-critical situations.

For Oracle, at a scale factor 0.1, total execution times without encryption go from 6,270ms to 11,860ms, with moderate resource consumption. While AES-128 and AES-256 have slightly lower minimum execution times (6,050 ms and 6,360 ms respectively), their average execution times are actually higher than those of the non-encrypted runs. AES-192 shows a more inconsistent performance with execution times ranging from 4,230ms to 15,380ms, regarding total execution time, with a

much higher disk time usage up to 29.89%, which could indicate inefficient resource allocation or bottlenecks in disk I/O.

At Scale Factor 1, Oracle shows some unexpected inconsistencies when compared to scale factor 0.1. However, some encryption algorithms outperform the no-encryption in terms of execution time. Without encryption total execution times range from 169,410ms to 293,030ms, which is better than the times produced by the encryption algorithms. AES-128 had the worst performance with a minimum total execution time of 986,750ms among the three test runs compared to AES-192 and AES-256 with a minimum execution time of 894,150 ms and 480,970 ms respectively. AES-256 shows the highest variability at Scale Factor 1, with a standard deviation of 472,856 ms, compared to 48,280 ms for AES-192 and 7,056 ms for AES-128 (calculated using the STDEV.S function across the 3 test runs). Although AES-192 performed better than AES-128 in terms of average execution time, its disk usage had a poorer performance, up to 135.80%, and it was less efficient overall. AES-256 had the best average execution but suffered from significant inconsistency, making AES-128 a more stable choice when choosing consistency of results.

At Scale Factor 10 the performance becomes more variable, especially with stronger encryption algorithms like AES-192 and AES-256. Non-encrypted total execution times range from 1,287,610ms to 1,446,090ms, AES-128 execution times range from 1,532,510ms to 1,555,030ms. However, AES-192 and AES-256 show spikes in their execution times, much higher than those of both AES-128 and no-encryption. AES-256 has the highest total execution time of 10,614,900ms. This can be due to high disk time usage going up to 61.73% and the greater number of encryption rounds required by AES-256's larger key size.

For MySQL, AES-256 is a balanced encryption algorithm when using a scale factor of 0.1. It has faster execution times, ranging from 33,310ms to 37,400ms, compared to non-encrypted total execution times, which range from 55,930ms to 97,690ms. AES-256 has a consistent CPU usage, between 40.4% and 42.5%, and low disk usage ranging from 7.45% to 7.93%, showing minimal dependence on disk operations. This combination of low total execution times and low disk usage makes AES-256 a strong choice for MySQL under smaller scale factors.

When using scale factor 1, AES-256 had slower execution times compared to non-encrypted results. This can be explained by high disk usage in some tests, reaching up to 95.53%. AES-256 had execution times ranging from 1,977,470ms to 2,135,410ms, higher than those in the unencrypted setup. While CPU usage stayed moderate, AES-256 performs worse than the non-encrypted tests, which reached up to 1,997,740ms. Disk time is a limiting factor causing AES-256 to perform worse under heavier workloads.

Overall, for SQL Server when assessing performance across different scale factors, AES-192 was most balanced, with a moderate variability and strong average execution times. AES-256 showed high variability, especially in higher scale factors. AES-192 was the most consistent, when assessing the data across the different scale

factors, with the lowest standard deviation among the three encryption algorithms in almost every scale factor except at scale factor 0.1. AES-192 did not lead in every category, but had consistent test runs with better average execution times, expect at Scale Factor 1 where AES-128 performed better. At scale factor 0.1, AES-192 had the highest standard deviation (1,360 ms) compared to AES-128 (247 ms) and AES-256 (887 ms).

For MySQL v8.0.38, AES-256 showed a more stable performance, though it's execution times stayed higher than the non-encrypted test runs, making AES-256 the best option when using lower scale factors.

Oracle, when using AES-128 and AES-256, handles the query processing under lower scale factors well, but AES-192 and AES-256 under scale factor 10 showed performance degradation, taking more execution time compared to AES-128 and non-encrypted results, making AES-192 the most balanced, offering encryption strength and reasonable performance in Oracle.

The MySQL system uses high disk usage because it depends on disk operations during query execution. SQL Server delivers faster execution times throughout various scale factors and encryption algorithms while maintaining lower disk usage. The CPU and RAM usage of both platforms appears normal. SQL Server adjusts memory usage, but MySQL maintains conservative CPU usage during encrypted database queries. SQL Server demonstrates superior readiness for demanding performance scenarios because it creates optimal execution plans for complex queries and elevated scale factors. The CPU's resources receive efficient management by MySQL.

5.2 Query Execution time per platform and factor

This section presents the results for each query, with tables specifically displaying the corresponding results for each database platform and scale factor. Additionally, it includes an analysis of these results to highlight trends, performance differences, and the implications for choosing the appropriate encryption algorithms.

5.2.1 Oracle Scale Factor 0.1

Table 5 and Figure 15 show the results for the executions in Oracle for the scale factor 0.1. This section analysed the performance of the AES encryption algorithms AES-128, AES-192, AES-256 compared to the non-encryption across 22 queries executed three times, while the grand total reflects all 66 executions combined.

The data revealed that AES-192 the encryption algorithm shows slightly better total execution time among the encryption algorithms with a lower total execution time than AES-128 and AES-256, though performance varies across queries. This is reflected in the variability in execution times, which AES-192 has the lowest variability (467 ms) in execution times compared to AES-256 (401 ms) and AES-128 (486 ms).

AES-192 ranges from 30ms to 2,990ms, AES-128 ranges from 70ms to 3,350ms, and AES-256 from 60ms to 2,540ms.

The performance and consistency of AES-192 comes from its balanced and efficient utilization of system resources. AES-192 shows a moderate use of RAM usage (25.15% to 27.80%) and CPU usage (40.63% to 57.17%) paired with a consistent higher use of Disk Time (23.95% to 29.89%). In contrast, AES-128 shows inefficiencies in resource utilization. Its RAM usage shows a high variation from 27.66% to 55.02%, while CPU usage ranges from 35.27% and 41.11%. This suboptimal allocation of computational resources contributes to slower execution times.

AES-256 while having some similarities with AES-192 in terms of execution time, shows a slower performance, totalling 27,050ms against AES-192's total execution time 26,710ms. AES-256 showed more consistent performance with a lower standard deviation (401 ms) compared to AES-192 (467 ms), indicating less variability across executions. While the total between AES-192 and AES-256 is not so far from each other, this inconsistency can be linked to the lack of effective disk time (1.28% to 3.43%) usage. The queries Q5, Q14, Q18, Q19, and Q20 shows higher variations and results not expected, for example, in Q14, AES-256 achieves a 22% faster execution time than AES-128 and 14% faster than AES-192. AES-256 achieves this by leveraging more efficient CPU usage.

The no-encryption results showed consistencies in performance, further supported by the lowest standard deviation (325 ms), showing the most stable execution times across all queries. The total execution time for no-encryption was 25,100ms, which is lower than both AES-192 (26,710ms), AES-256 (27,050ms), and AES-128 (27,610ms). This outcome is expected, as when without encryption, the executions normally results in faster times. While it is generally expected that no-encryption would perform better than encrypted algorithms due to the absence of encryption overhead, the small percentage gap between no-encryption and the encrypted algorithms can be explained due to the AES-NI support, as shown in Appendix D, to CPU usage, reducing the overhead and making the encryption and decryption processes when querying as efficient as a non-encryption executions, though not outperforming them.

Table 5 - Oracle Scale Factor 0.1 results

Query Number	**Execution Number**	**AES-128 (ms)**	**AES-192 (ms)**	**AES-256 (ms)**	**No-encryption (ms)**
Q1	1	350	370	360	370
	2	330	480	330	350
	3	340	160	340	350
	Average	*340*	*337*	*343*	*357*
	Stddev	*10*	*163*	*15*	*12*
Q2	1	650	670	670	180
	2	180	140	140	50

Query Number	**Execution Number**	**AES-128 (ms)**	**AES-192 (ms)**	**AES-256 (ms)**	**No-encryp-tion (ms)**
	3	130	60	130	60
	Average	*320*	*290*	*313*	*97*
	Stddev	*287*	*332*	*309*	*72*
Q3	1	530	580	510	560
	2	120	310	270	270
	3	270	150	280	270
	Average	*307*	*347*	*353*	*367*
	Stddev	*207*	*217*	*136*	*167*
Q4	1	270	290	270	260
	2	260	250	240	260
	3	240	120	320	250
	Average	*257*	*220*	*277*	*257*
	Stddev	*15*	*89*	*40*	*6*
Q5	1	2,240	2,150	2,020	2,260
	2	180	320	320	310
	3	310	140	310	320
	Average	*910*	*870*	*883*	*963*
	Stddev	*1,154*	*1,112*	*984*	*1,123*
Q6	1	170	170	160	180
	2	170	160	150	190
	3	150	160	170	190
	Average	*163*	*163*	*160*	*187*
	Stddev	*12*	*6*	*10*	*6*
Q7	1	730	780	710	750
	2	370	340	320	400
	3	320	160	400	330
	Average	*473*	*427*	*477*	*493*
	Stddev	*224*	*319*	*206*	*225*
Q8	1	390	410	410	440
	2	300	260	250	270
	3	270	120	250	270
	Average	*320*	*263*	*303*	*327*
	Stddev	*62*	*145*	*92*	*98*
Q9	1	1,100	1,210	1,090	670
	2	570	570	550	560
	3	550	290	600	600
	Average	*740*	*690*	*747*	*610*
	Stddev	*312*	*472*	*298*	*56*
Q10	1	730	750	690	860
	2	330	290	250	260

Query Number	**Execution Number**	**AES-128 (ms)**	**AES-192 (ms)**	**AES-256 (ms)**	**No-encryp-tion (ms)**
	3	260	180	270	260
	Average	*440*	*407*	*403*	*460*
	Stddev	*254*	*302*	*248*	*346*
Q11	1	120	130	120	100
	2	110	100	100	100
	3	100	90	120	100
	Average	*110*	*107*	*113*	*100*
	Stddev	*10*	*21*	*12*	*0*
Q12	1	360	400	350	360
	2	300	290	270	260
	3	280	130	310	280
	Average	*313*	*273*	*310*	*300*
	Stddev	*42*	*136*	*40*	*53*
Q13	1	380	400	390	420
	2	280	320	270	370
	3	280	130	300	290
	Average	*313*	*283*	*320*	*360*
	Stddev	*58*	*139*	*62*	*66*
Q14	1	3,350	2,990	2,540	240
	2	160	170	160	170
	3	160	170	160	170
	Average	*1,223*	*1,110*	*953*	*193*
	Stddev	*1,842*	*1,628*	*1,374*	*40*
Q15	1	220	240	220	230
	2	220	230	210	190
	3	190	90	190	210
	Average	*210*	*187*	*207*	*210*
	Stddev	*17*	*84*	*15*	*20*
Q16	1	190	200	180	190
	2	130	130	130	120
	3	130	130	130	130
	Average	*150*	*153*	*147*	*147*
	Stddev	*35*	*40*	*29*	*38*
Q17	1	200	190	200	220
	2	180	180	160	360
	3	160	170	160	200
	Average	*180*	*180*	*173*	*260*
	Stddev	*20*	*10*	*23*	*87*
Q18	1	700	700	730	690
	2	190	440	430	420

Query Number	Execution Number	AES-128 (ms)	AES-192 (ms)	AES-256 (ms)	No-encryption (ms)
	3	410	200	430	440
	Average	*433*	*447*	*530*	*517*
	Stddev	*256*	*250*	*173*	*150*
Q19	1	850	840	840	1,060
	2	580	570	570	560
	3	550	740	550	560
	Average	*660*	*717*	*653*	*727*
	Stddev	*165*	*137*	*162*	*289*
Q20	1	660	840	640	750
	2	720	640	640	680
	3	220	100	210	220
	Average	*533*	*527*	*497*	*550*
	Stddev	*273*	*383*	*248*	*288*
Q21	1	920	940	910	950
	2	540	800	700	720
	3	670	710	670	710
	Average	*710*	*817*	*760*	*793*
	Stddev	*193*	*116*	*131*	*136*
Q22	1	120	130	120	120
	2	110	110	100	100
	3	60	30	60	60
	Average	*97*	*90*	*93*	*93*
	Stddev	*32*	*53*	*31*	*31*
Total	**All Executions**	**27,610**	**26,710**	**27,050**	**25,100**
Total Stddev	**All Executions**	**486**	**467**	**401**	**325**

AES-128 was the encryption algorithm with the poorest performance, the broader range shows occasional inefficiencies, especially for Q14 as shown in Figure 15. Q14 calculates the revenue percentage from promotional items for a specific month, accessing the Lineitem and Part tables. Lineitem is one of the largest tables, and the query involves a join and a condition aggregation which adds complexity, requiring significant processing.. To improve the performance of this query, views and indexed views can be created, which is going to reduce the need to repeatedly calculate the SUM(L_QUANTITY), allowing the query to fetch aggregated data directly from

the view and reducing the computation complexity when executing this query, especially when encryption is involved.

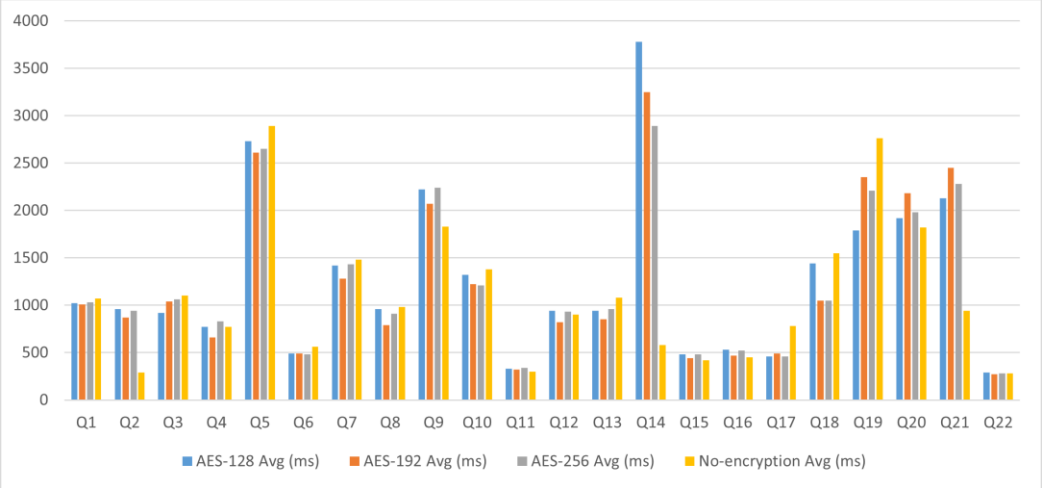


Figure 15 - Average query execution times (in ms) for AES-128, AES-192, AES-256, and No-encryption algorithms across 22 queries in Oracle – Scale Factor 0.1

5.2.2 Oracle Scale Factor 1

Table 6 and Figure 16 show the results for the executions in Oracle for the scale factor 1. This section analysed the performance of the AES encryption algorithms AES-128, AES-192, AES-256 compared to the non-encryption across 22 queries executed three times, while the grand total reflects all 66 executions combined.

The results show as expected no-encryption configuration to deliver the best performance compared to encrypted test cases. No-encryption delivered the best performance with an average total execution time of 210,963.33ms and the lowest standard deviation of 17,077 compared to the encryption algorithms. AES-256 had the best average execution time among the encryption algorithms, averaging around 754,733.33ms of execution time with a standard deviation of 131,863 ms. AES-192 and AES-128 were slower than AES-256 with the respective times of 942,346.67 ms and 994,026.67 ms in average total execution time.

Among the encryption algorithms AES-128 and AES-192 were slower than AES-256 and no-encryption results. Although AES-256 had the best average performance, AES-192 showed consistency with a standard deviation of 24,271 ms, compared to 131,863 ms for AES-256 and 135,729 ms for AES-128. These findings reinforce the benefits of using AES-NI hardware-accelerated encryption technology which can manage and mitigate the performance penalties, allowing to achieve acceptable performance levels.

Table 6 – Oracle Scale Factor 1 results

Query Number	**Execution Number**	**AES-128 (ms)**	**AES-192 (ms)**	**AES-256 (ms)**	**No-encryption (ms)**
Q1	1	3,650	40,950	1,660	1,620
	2	3,860	39,240	1,690	1,660
	3	3,720	37,360	1,830	1,580
	Average	*3,743*	*39,183*	*1,727*	*1,620*
	Stdev	*107*	*1,796*	*91*	*40*
Q2	1	3,100	49,970	20,630	27,510
	2	3,730	35,740	1,720	1,540
	3	3,660	490	5,050	4,980
	Average	*3,497*	*28,733*	*9,133*	*11,343*
	Stdev	*345*	*25,473*	*10,095*	*14,106*
Q3	1	14,350	68,310	7,220	6,640
	2	14,440	72,630	7,110	6,690
	3	14,480	70,340	6,740	6,730
	Average	*14,423*	*70,427*	*7,023*	*6,687*
	Stdev	*67*	*2,161*	*251*	*45*
Q4	1	2,670	41,020	1,230	1,180
	2	2,520	38,460	1,240	1,210
	3	2,660	39,640	1,270	1,190
	Average	*2,617*	*39,707*	*1,247*	*1,193*
	Stdev	*84*	*1,281*	*21*	*15*
Q5	1	16,910	51,070	10,090	9,080
	2	17,840	50,540	8,760	9,200
	3	16,930	47,310	8,330	8,040
	Average	*17,227*	*49,640*	*9,060*	*8,773*
	Stdev	*531*	*2,035*	*918*	*638*
Q6	1	1,370	29,200	2,510	2,180
	2	1,660	32,100	780	740
	3	1,610	29,190	770	820
	Average	*1,547*	*30,163*	*1,353*	*1,247*
	Stdev	*155*	*1,677*	*1,002*	*809*
Q7	1	4,620	47,910	2,350	2,050
	2	4,690	48,270	2,190	2,290
	3	4,600	44,920	2,140	2,190
	Average	*4,637*	*47,033*	*2,227*	*2,177*
	Stddev	*47*	*1,839*	*110*	*121*
Q8	1	44,280	47,480	22,680	26,850
	2	45,240	40,020	21,130	21,550
	3	44,700	40,430	21,490	21,700
	Average	*44,740*	*42,643*	*21,767*	*23,367*

Query Number	**Execution Number**	**AES-128 (ms)**	**AES-192 (ms)**	**AES-256 (ms)**	**No-encryption (ms)**
	Stddev	*481*	*4,194*	*811*	*3,018*
Q9	1	126,080	63,980	140,890	91,420
	2	120,500	63,810	59,460	61,210
	3	123,110	67,460	60,060	60,780
	Average	*123,230*	*65,083*	*86,803*	*71,137*
	Stddev	*2,792*	*2,060*	*46,841*	*17,567*
Q10	1	10,510	40,430	6,580	5,710
	2	10,680	41,610	5,380	5,970
	3	10,450	40,340	5,180	4,920
	Average	*10,547*	*40,793*	*5,713*	*5,533*
	Stddev	*119*	*709*	*757*	*547*
Q11	1	8,440	6,410	3,710	3,730
	2	8,250	6,590	3,700	3,410
	3	8,430	6,860	3,650	3,340
	Average	*8,373*	*6,620*	*3,687*	*3,493*
	Stddev	*107*	*226*	*32*	*208*
Q12	1	2,800	43,780	1,430	1,330
	2	2,970	45,290	1,380	1,420
	3	2,990	42,630	1,390	1,380
	Average	*2,920*	*43,900*	*1,400*	*1,377*
	Stddev	*104*	*1,334*	*26*	*45*
Q13	1	1,930	26,280	1,680	1,270
	2	2,670	26,270	1,180	1,420
	3	2,540	27,870	1,200	1,160
	Average	*2,380*	*26,807*	*1,353*	*1,283*
	Stddev	*395*	*921*	*283*	*131*
Q14	1	2,660	31,700	5,860	6,580
	2	2,760	30,560	1,260	1,160
	3	2,710	28,640	1,310	1,200
	Average	*2,710*	*30,300*	*2,810*	*2,980*
	Stddev	*50*	*1,546*	*2,641*	*3,118*
Q15	1	2,090	29,930	980	940
	2	2,140	32,440	990	1,050
	3	2,180	32,350	990	1,020
	Average	*2,137*	*31,573*	*987*	*1,003*
	Stddev	*45*	*1,424*	*6*	*57*
Q16	1	2,660	17,210	1,400	1,570
	2	2,530	11,980	1,180	1,350
	3	2,940	2,910	1,180	1,180
	Average	*2,710*	*10,700*	*1,253*	*1,367*

Query Number	Execution Number	AES-128 (ms)	AES-192 (ms)	AES-256 (ms)	No-encryption (ms)
	Stddev	*210*	*7,235*	*127*	*196*
Q17	1	47,710	30,080	24,380	18,790
	2	47,120	41,370	22,240	20,810
	3	44,150	30,660	23,140	20,400
	Average	*46,327*	*34,037*	*23,253*	*20,000*
	Stddev	*1,908*	*6,357*	*1,074*	*1,068*
Q18	1	30,450	102,420	41,830	68,290
	2	27,160	105,320	14,960	13,820
	3	31,590	98,150	14,620	13,800
	Average	*29,733*	*101,963*	*23,803*	*31,970*
	Stddev	*2,300*	*3,607*	*15,612*	*31,454*
Q19	1	7,230	38,360	3,540	3,410
	2	7,200	36,000	3,290	3,390
	3	6,920	35,420	3,320	3,220
	Average	*7,117*	*36,593*	*3,383*	*3,340*
	Stddev	*171*	*1,557*	*137*	*104*
Q20	1	659,320	77,640	996,020	9,350
	2	649,510	44,540	318,750	6,410
	3	656,270	76,070	313,410	6,170
	Average	*655,033*	*66,083*	*542,727*	*7,310*
	Stddev	*5,021*	*18,674*	*392,573*	*1,771*
Q21	1	7,380	96,720	3,670	3,230
	2	8,640	89,930	3,780	3,840
	3	7,280	86,060	3,400	3,310
	Average	*7,767*	*90,903*	*3,617*	*3,460*
	Stddev	*758*	*5,396*	*196*	*332*
Q22	1	630	9,860	400	300
	2	640	9,470	320	310
	3	570	9,050	500	300
	Average	*613*	*9,460*	*407*	*303*
	Stddev	*38*	*405*	*90*	*6*
Total	**All Executions**	**2,982,080**	**2,827,040**	**2,264,200**	**632,890**
Total Stddev	**All Executions**	**135,729**	**24,271**	**131,863**	**17,077**

AES-256 is more consistent than the other encryption algorithm key sizes, performing better than AES-128 and AES192, while in Q20 there was a spike which affected AES-128 and AES-256, this query shows multiple table joins up to four tables, Partsupp, Supplier, Nation, and Lineitem. It also shows a subquery which increases the performance cost. AES-256 is the best choice for the scale factor 1, it showed a

better performance in almost all executions and provides the stronger encryption, having the less spike and variable results while looking at Figure 16.

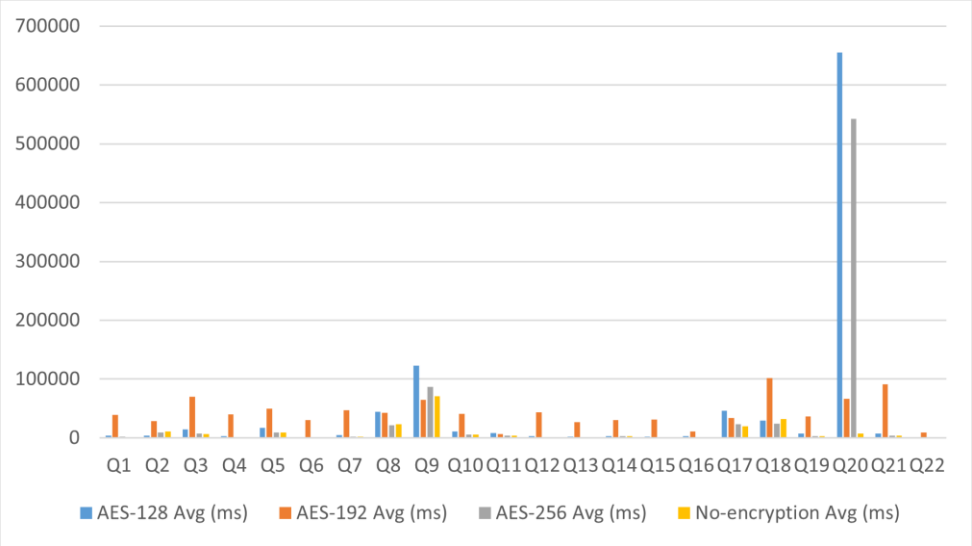


Figure 16 - Average query execution times (in ms) for AES-128, AES-192, AES-256, and No-encryption algorithms across 22 queries in Oracle – Scale Factor 1

5.2.3 Oracle Scale Factor 10

The results for Oracle executions at scale factor 10 appear in Table 7 and Figure 17. This section analysed the performance of AES encryption algorithms AES-128, AES-192 and AES-256 against no-encryption across 22 queries executed three times, while the grand total reflects all 66 executions combined.

The total execution times for all 22 queries show differences between the algorithms. AES-128 has an efficient performance, completing in 4,622,030ms. AES-192 demonstrates a higher total time of 5,630,370ms, while AES-256 incurs a significantly more significant overhead with 24,373,640ms. No-encryption shows the fastest total execution time, completing in 4,068,940ms. The advantage can be attributed to AES-NI hardware acceleration which optimizes CPU and I/O usage during encryption.

From Q1 to Q5, AES-128 and AES-192 show similar performance, with AES-128 generally providing faster execution times. Q1, AES-128 had an average completed time in 79,533.33ms compared to AES-192's 81,440ms. Again, AES-128 outperforms AES-192 in Q5, with a total execution time of 90,023.33ms compared to AES-192's 337,386.67ms. At the same time, AES-256 has higher execution times in this range, particularly in Q3, where its execution time is 856,196.67ms compared to AES-128's 99,890ms.

From Q6 to Q10, AES-128 is the most efficient encryption algorithm for most queries, with consistent performance over AES-192 and AES-256. For example, in Q7, AES-128 completed in 81,950ms, slightly slower than AES-192's 79,120ms, while AES-256 followed behind with 507,837ms. No-encryption performs efficiently in

these queries but shows variances due to memory inefficiencies. AES-256 continues to show a considerable performance lag, especially in Q9, with an execution time of 1,726,767ms compared to AES-128's 122,620ms.

From Q11 to Q15, AES-192 efficiently handled complex data querying. For instance, in Q13, AES-192 achieved 87,760ms, outperforming AES-128's 114,190ms. However, in Q14, AES-192's 163,970ms slightly underperformed compared to AES-128's 160,430ms being slightly faster. AES-256, while slower, provides strong encryption and reasonable performance, especially for queries involving large datasets.

From Q16 to Q22, AES-192 continue to perform well across most queries. In Q16, AES-192 performed better than AES-128, completing in 22,600ms compared to AES-128's 25,290ms, indicating efficiency in handling smaller workloads. No-encryption outperforms encryption algorithms in some queries, such as Q20 and Q21, where it had the fastest total execution times at 204,560ms and 417,570ms. AES-256 shows a high overhead in this range, with significant overhead performance in some queries such as in Q18, where it got 4,171,380ms, having an higher total execution time of the 3 executions compared to AES-128 (517,520ms) and AES-192 (599,650ms).

Overall, AES-128 is the most efficient encryption algorithm for scale factor 10, it has both the lowest total execution time (4,622,030 ms) and the lowest standard deviation (40,482 ms) among the encryption algorithms. AES-192 is very competitive with AES-128, having close results with each other, making it a valid choice. AES-256, with higher computational demand, provides strong security and has a higher total execution time which is also expected. The no-encryption algorithm outperforms the encryption algorithms' results as expected. These findings highlight the importance of having higher encryption security and maintaining system performance, with AES-128 and AES-192 offering fine solutions for most database operations at scale factor 10.

Table 7 - Oracle Scale Factor 10 results

Query Number	**Execution Number**	**AES-128 (ms)**	**AES-192 (ms)**	**AES-256 (ms)**	**No-encryption (ms)**
Q1	1	81,430	90,470	74,850	88,700
	2	78,510	76,880	75,430	77,610
	3	78,660	76,970	78,780	75,390
	Average	*79,533*	*81,440*	*76,353*	*80,567*
	Stdev	*1,644*	*7,820*	*2,121*	*7,131*
Q2	1	38,050	16,700	333,990	40,740
	2	28,650	35,580	6,240	780
	3	1,020	710	1,020	1,050
	Average	*22,573*	*17,663*	*113,750*	*14,190*
	Stdev	*19,248*	*17,455*	*190,751*	*22,993*

Query Number	**Execution Number**	**AES-128 (ms)**	**AES-192 (ms)**	**AES-256 (ms)**	**No-encryption (ms)**
Q3	1	102,080	146,650	1,318,560	135,930
	2	97,180	139,040	1,153,320	71,400
	3	100,410	116,960	96,710	122,170
	Average	*99,890*	*134,217*	*856,197*	*109,833*
	Stdev	*2,491*	*15,421*	*662,904*	*33,988*
Q4	1	78,800	79,860	70,280	62,120
	2	75,170	64,050	90,050	62,780
	3	79,950	65,940	81,280	64,640
	Average	*77,973*	*69,950*	*80,537*	*63,180*
	Stdev	*2,495*	*8,634*	*9,906*	*1,307*
Q5	1	88,990	362,420	148,390	79,140
	2	89,820	312,800	145,700	79,410
	3	91,260	336,940	90,030	72,630
	Average	*90,023*	*337,387*	*128,040*	*77,060*
	Stdev	*1,149*	*24,813*	*32,945*	*3,839*
Q6	1	50,150	60,480	52,980	35,670
	2	52,500	53,680	48,010	49,190
	3	53,070	51,550	51,190	46,930
	Average	*51,907*	*55,237*	*50,727*	*43,930*
	Stdev	*1,548*	*4,664*	*2,517*	*7,242*
Q7	1	82,460	88,790	405,750	72,550
	2	81,890	82,260	425,740	68,730
	3	81,500	66,310	692,020	72,280
	Average	*81,950*	*79,120*	*507,837*	*71,187*
	Stdev	*483*	*11,564*	*159,820*	*2,132*
Q8	1	73,030	62,980	934,010	68,670
	2	74,370	68,230	645,720	61,990
	3	72,240	63,260	882,630	60,720
	Average	*73,213*	*64,823*	*820,787*	*63,793*
	Stdev	*1,077*	*2,954*	*153,773*	*4,271*
Q9	1	119,370	126,920	1,738,960	102,790
	2	123,590	111,690	1,734,000	105,070
	3	124,900	106,170	1,707,340	104,290
	Average	*122,620*	*114,927*	*1,726,767*	*104,050*
	Stdev	*2,890*	*10,747*	*17,006*	*1,159*
Q10	1	68,070	111,850	1,351,370	56,650
	2	69,920	95,770	1,276,100	58,790
	3	68,810	93,970	69,190	61,570
	Average	*68,933*	*100,530*	*898,887*	*59,003*
	Stdev	*931*	*9,845*	*719,523*	*2,467*

Query Number	**Execution Number**	**AES-128 (ms)**	**AES-192 (ms)**	**AES-256 (ms)**	**No-encryption (ms)**
Q11	1	12,980	10,370	146,800	10,230
	2	13,270	11,050	316,250	10,820
	3	13,970	13,420	86,870	10,640
	Average	*13,407*	*11,613*	*183,307*	*10,563*
	Stdev	*509*	*1,601*	*118,968*	*302*
Q12	1	72,710	82,100	64,170	68,130
	2	74,520	67,800	84,150	61,560
	3	78,390	68,710	72,750	58,510
	Average	*75,207*	*72,870*	*73,690*	*62,733*
	Stdev	*2,902*	*8,006*	*10,023*	*4,916*
Q13	1	38,120	38,520	37,830	45,360
	2	38,670	24,550	62,850	26,640
	3	37,400	24,690	51,370	27,920
	Average	*38,063*	*29,253*	*50,683*	*33,307*
	Stdev	*637*	*8,025*	*12,524*	*10,458*
Q14	1	53,610	59,360	84,390	53,360
	2	52,340	53,830	64,120	50,250
	3	54,480	50,780	66,020	53,440
	Average	*53,477*	*54,657*	*71,510*	*52,350*
	Stdev	*1,076*	*4,349*	*11,195*	*1,819*
Q15	1	59,460	64,670	56,350	61,180
	2	58,840	60,400	54,570	57,520
	3	57,880	57,480	53,720	55,340
	Average	*58,727*	*60,850*	*54,880*	*58,013*
	Stdev	*796*	*3,616*	*1,342*	*2,951*
Q16	1	9,560	8,900	30,410	11,370
	2	10,130	8,350	7,100	5,180
	3	5,600	5,350	6,440	5,410
	Average	*8,430*	*7,533*	*14,650*	*7,320*
	Stdev	*2,467*	*1,911*	*13,653*	*3,509*
Q17	1	48,880	64,500	616,730	34,640
	2	51,200	58,390	638,410	55,520
	3	55,360	54,340	49,510	51,400
	Average	*51,813*	*59,077*	*434,883*	*47,187*
	Stdev	*3,283*	*5,115*	*333,919*	*11,059*
Q18	1	168,900	234,880	2,588,320	130,910
	2	174,050	179,330	1,404,840	116,110
	3	174,570	185,440	178,220	114,110
	Average	*172,507*	*199,883*	*1,390,460*	*120,377*
	Stdev	*3,134*	*30,462*	*1,205,114*	*9,177*

Query Number	**Execution Number**	**AES-128 (ms)**	**AES-192 (ms)**	**AES-256 (ms)**	**No-encryption (ms)**
Q19	1	68,000	106,180	107,320	61,830
	2	67,440	94,030	115,470	68,430
	3	68,530	89,660	66,280	65,280
	Average	*67,990*	*96,623*	*96,357*	*65,180*
	Stdev	*545*	*8,560*	*26,364*	*3,301*
Q20	1	65,960	81,900	207,860	76,670
	2	73,060	66,210	404,650	61,940
	3	70,000	66,660	262,670	65,950
	Average	*69,673*	*71,590*	*291,727*	*68,187*
	Stdev	*3,561*	*8,932*	*101,562*	*7,615*
Q21	1	137,300	162,670	231,590	146,550
	2	157,890	142,280	183,720	131,690
	3	152,270	140,280	147,280	139,330
	Average	*149,153*	*148,410*	*187,530*	*139,190*
	Stdev	*10,643*	*12,390*	*42,284*	*7,431*
Q22	1	14,600	14,940	13,990	2,900
	2	12,020	5,970	15,250	6,200
	3	14,220	6,500	15,730	6,240
	Average	*13,613*	*9,137*	*14,990*	*5,113*
	Stdev	*1,393*	*5,033*	*899*	*1,917*
Total	**All Executions**	**4,622,030**	**5,630,370**	**24,373,640**	**4,068,940**
Total Stddev	**All Executions**	**40,482**	**72,471**	**545,955**	**35,882**

In Figure 17 it is possible to see some outliers such as Q3, Q7, Q8, Q9, Q10, Q11, Q17 and Q18 for AES-256, which shows a high computational demand as said before. This Figure 17 also shows that AES-128 are very close to each other in terms of results, although AES-192 has one significant outlier compared to AES-128 which is at Q5, query 5 involves six tables, one of them being the Lineitem which is the largest table among the eight tables, knowing this it's interesting to see that AES-192 has a bad performance compared to AES-128 as expected and a worst performance compared to AES-256 which is not expected, remembering that AES-128 takes 10 rounds to encrypt, AES-192 takes 12 rounds to encrypt and AES-256 takes 14 rounds to encrypt, which is possible to see while scaling the size of the database to query.

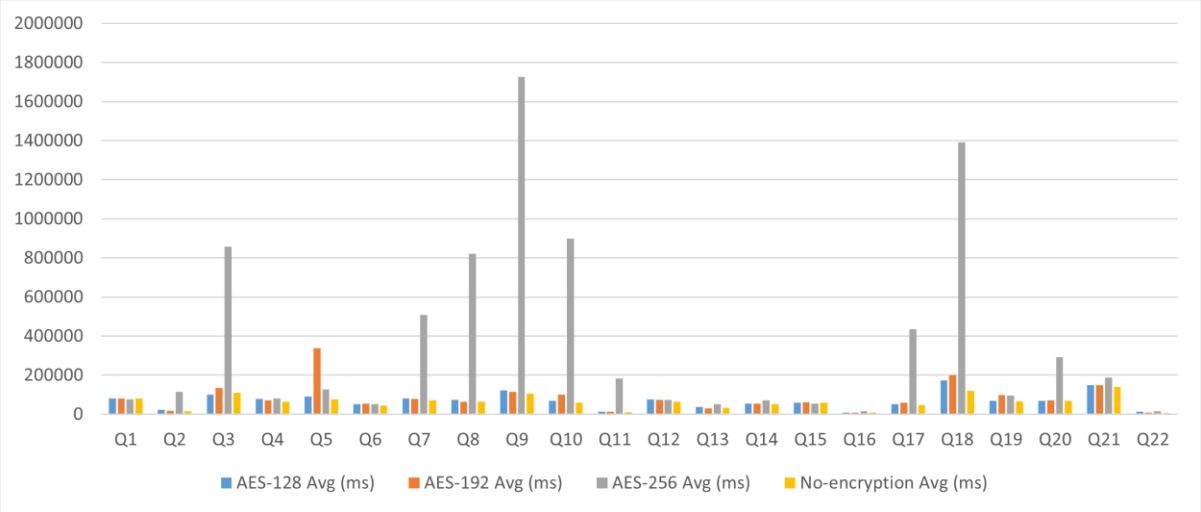


Figure 17 - Average query execution times (in ms) for AES-128, AES-192, AES-256, and No-encryption algorithms across 22 queries in Oracle – Scale Factor 10

5.2.4 MySQL Scale Factor 0.1

The results from MySQL executions for scale factor 0.1 appear in Table 8 and Figure 18. This section examined how AES-256 encryption performed relative to no-encryption during 22 query executions that were run three times, while the grand total reflects all 66 executions combined.

At scale factor 0.1, AES-256 encryption in MySQL had a reduced execution time combination of the 22 queries from 217,350ms (no-encryption) to 104,610ms (AES-256), stating that the encrypted workload ran in under half the time of the no-encryption. AES-256 demonstrates better efficiency than no-encryption throughout most execution cases. The processing times of no-encryption show significant fluctuations throughout the evaluation. When evaluating individual queries, runtimes without encryption ranged from 1,920ms (Q2) to 29,320ms (Q9), while encrypted runs ranged from 1,270ms (Q5) to 9,070ms (Q18). On average, each query completed in 4,755ms under AES-256 compared to 9,880ms without encryption, confirming that AES-256 takes less than half to execute compared to no-encryption making the best choice to smaller factors. Additionally, the overall performance of AES-256 was more consistent, with a total standard deviation of 1,396 ms, compared to 4,258 ms for no-encryption.

Evaluating each of the three executions separately, run times without encryption had a total of 97,690ms, 63,730ms, and 55,930ms for runs 1,2 and 3 respectively. With AES-256, these dropped to 37,400ms, 33,900ms, and 33,310ms, corresponding to a reduction of 61.8%, 46.9%, and 40.4%. In average it took for each execution of no-encryption 72,450ms to execute and for AES-256 an average of 34,870ms, a 51.9% decrease in execution time per run.

These difference between AES-256 and no-encryption may relate to internal query handling or engine-level optimizations such as AES-NI hardware acceleration that can contribute to better CPU and I/O resources or memory management.

Table 8 - MySQL Scale Factor 0.1 results

Query Number	**Execution Number**	**AES-256 (ms)**	**No-encryption (ms)**
Q1	1	3,530	3,480
	2	1,190	1,170
	3	3,100	3,430
	Average	*2,607*	*2,693*
	Stdev	*1,246*	*1,319*
Q2	1	180	210
	2	680	990
	3	410	720
	Average	*423*	*640*
	Stdev	*250*	*396*
Q3	1	740	740
	2	880	2,070
	3	930	570
	Average	*850*	*1,127*
	Stdev	*98*	*821*
Q4	1	1,350	8,490
	2	820	830
	3	910	710
	Average	*1,027*	*3,343*
	Stdev	*284*	*4,458*
Q5	1	920	1,250
	2	330	1,150
	3	1,480	2,100
	Average	*910*	*1,500*
	Stdev	*575*	*522*
Q6	1	1,520	1,480
	2	840	3,170
	3	150	510
	Average	*837*	*1,720*
	Stdev	*685*	*1,346*
Q7	1	750	490
	2	3,990	5,670
	3	730	1,800
	Average	*1,823*	*2,653*
	Stdev	*1,876*	*2,693*
Q8	1	2,230	4,140

Query Number	Execution Number	AES-256 (ms)	No-encryption (ms)
	2	660	2,690
	3	200	380
	Average	*1,030*	*2,403*
	Stdev	*1,064*	*1,896*
Q9	1	4,860	26,820
	2	1,400	1,300
	3	1,240	1,200
	Average	*2,500*	*9,773*
	Stdev	*2,045*	*14,763*
Q10	1	1,300	2,940
	2	130	360
	3	420	410
	Average	*617*	*1,237*
	Stdev	*609*	*1,475*
Q11	1	230	1,020
	2	180	1,020
	3	1,650	1,640
	Average	*687*	*1,227*
	Stdev	*835*	*358*
Q12	1	1,540	1,170
	2	590	1,000
	3	5,490	11,970
	Average	*2,540*	*4,713*
	Stdev	*2,599*	*6,285*
Q13	1	1,980	2,160
	2	1,560	3,400
	3	2,150	1,870
	Average	*1,897*	*2,477*
	Stdev	*304*	*813*
Q14	1	1,250	4,460
	2	4,910	9,100
	3	1,100	1,100
	Average	*2,420*	*4,887*
	Stdev	*2,158*	*4,017*
Q15	1	3,830	11,460
	2	1,740	1,660
	3	990	1,330
	Average	*2,187*	*4,817*
	Stdev	*1,472*	*5,756*
Q16	1	1,180	6,060
	2	2,970	4,750

Query Number	Execution Number	AES-256 (ms)	No-encryption (ms)
	3	1,910	3,500
	Average	*2,020*	*4,770*
	Stdev	*900*	*1,280*
Q17	1	320	310
	2	1,240	3,840
	3	1,390	2,390
	Average	*983*	*2,180*
	Stdev	*579*	*1,774*
Q18	1	3,090	9,380
	2	1,450	4,500
	3	4,530	13,130
	Average	*3,023*	*9,003*
	Stdev	*1,541*	*4,327*
Q19	1	320	1,710
	2	4,970	11,420
	3	780	1,000
	Average	*2,023*	*4,710*
	Stdev	*2,562*	*5,822*
Q20	1	1,390	3,570
	2	540	510
	3	800	1,660
	Average	*910*	*1,913*
	Stdev	*436*	*1,546*
Q21	1	4,770	6,220
	2	330	610
	3	2,820	4,210
	Average	*2,640*	*3,680*
	Stdev	*2,225*	*2,842*
Q22	1	120	130
	2	2,500	2,520
	3	130	300
	Average	*917*	*983*
	Stdev	*1,371*	*1,334*
Total	**All Executions**	**104,610**	**217,350**
Total Stddev	**All Executions**	**1,396**	**4,258**

In Figure 18, it is possible to see that AES256 performed better than No-encryption in every single query, showing that AES256 is the best choice compared to no-encryption at scale factor 0.1. The average speedup is substantial in many queries, especially in Q4 being 88% faster, Q9 being 74% faster, and Q18 being 66% faster. Many of the possibilities to answer this overperformance against no-encryption is

that: can be because of the scale factor being so small and the system doesn't have control over the results when running multiple times, which can happen in some of the runs the no-encryption to get better results, caching behavior after the cold start, which encrypted reads may benefit from encryption overhead reducing the I/O time. AES-256 appears to be a viable and advantageous choice for smaller-scale factors in MySQL and still offers security, protecting sensitive information and in many cases improving query speed.

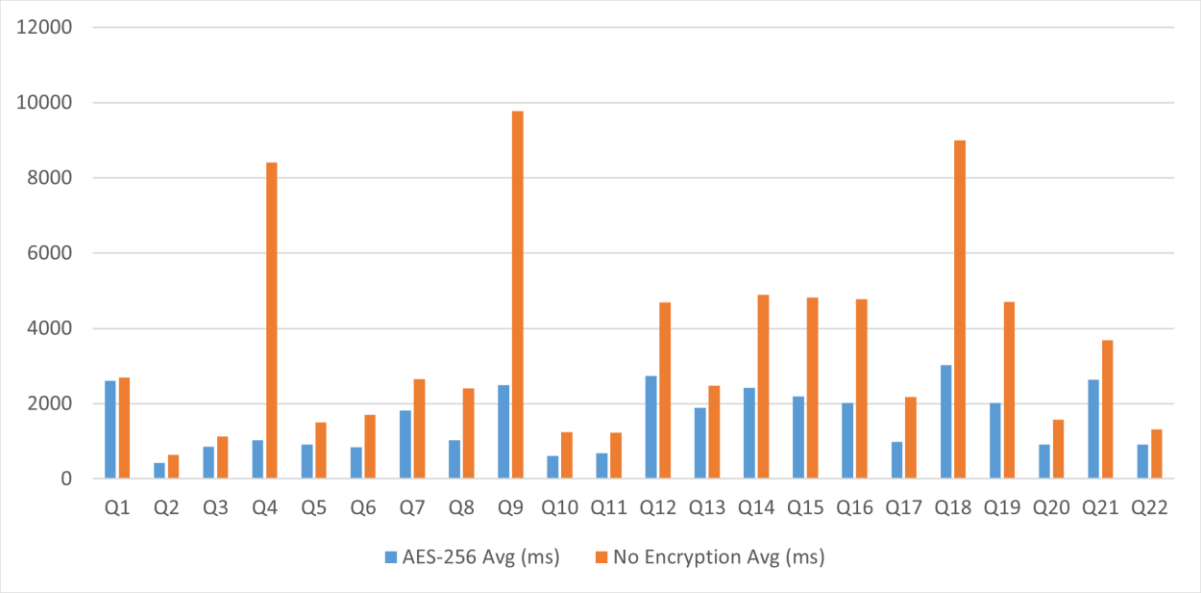


Figure 18 - Average query execution times (in ms) for AES-256, and No-encryption algorithms across 22 queries in MySQL – Scale Factor 0.1

5.2.5 MySQL Scale Factor 1

Table 9 and Figure 19 show the results of the executions in MySQL for scale factor 1. In this section, the performance of the AES encryption algorithm AES-256 is analyzed compared to no-encryption across 22 queries, with each executed three times, while the grand total reflects all 66 executions combined.

Across all runs, no encrypted took 5,851,760ms to complete, where AES-256 took 6,098,700ms, indicating that AES-256 takes 1.04 times to query all the runs, at this scale factor. Encryption incurs in a 4.2% overhead on total to process the entire set of runs in encryption compared to no encrypted set. Despite this slight overhead, both configurations showed similar variability, with a total standard deviation of 145,199 ms for AES-256 and 142,378 ms for no-encryption.

The analysis proves that AES-256 and no-encryption have similar average execution times, and both methods show variability across various test cases. AES-256 takes from 1,010ms to 538,320ms, while no-encryption takes from 1,920ms to 595,430ms. These variations indicate the presence of outliers that heavily influence the overall performances of both methods.

Under No-encryption, we see very high variability in the running times of tests, likely due to system bottlenecks or inefficiencies in handling unencrypted data, especially during I/O operations memory management. This is especially evident in the elevated standard deviations seen in Q2 (142,085 ms), Q9 (328,609 ms), and Q16 (282,171 ms), indicating inconsistent system response times. In certain test cases like execution one of Q9 and execution two of Q16, non-encryption outcomes show a considerable delay, making the overall average execution time go wrong. Query 9 performs complex joins across multiple tables, such as Part, Orders, Lineitem, Supplier, Partsupp, and Nation. A view can help optimise Query 9, precomputing some complex joins or subqueries. Query 16 does not use Lineitem, which has more data stored than the other tables, but uses nested queries, which can lead to high resource utilization. Views can make it simple to use query 16, storing complex filters, which can then be reused and useful for optimizing repeat queries with similar constraints.

AES-256, on the other hand, is found to be slower than no-encryption results, although having a small total overhead.

Table 9 - MySQL Scale Factor 1 results

Query Number	**Execution Number**	**AES-256 (ms)**	**No-encryption (ms)**
Q1	1	46,410	44,370
	2	21,660	20,660
	3	44,570	43,270
	Average	*37,547*	*36,100*
	Stdev	*13,789*	*13,383*
Q2	1	1,010	1,920
	2	279,280	251,930
	3	9,050	9,940
	Average	*96,447*	*87,930*
	Stdev	*158,389*	*142,085*
Q3	1	260,010	247,190
	2	18,440	19,010
	3	32,870	34,860
	Average	*103,773*	*100,353*
	Stdev	*135,497*	*127,411*
Q4	1	33,790	28,950
	2	41,650	44,200
	3	105,910	103,140
	Average	*60,450*	*58,763*
	Stdev	*39,565*	*39,180*
Q5	1	33,910	31,280
	2	3,760	3,750
	3	271,050	253,820
	Average	*102,907*	*96,283*

Query Number	Execution Number	AES-256 (ms)	No-encryption (ms)
	Stdev	*146,395*	*137,123*
Q6	1	21,050	20,770
	2	37,500	33,510
	3	14,940	13,760
	Average	*24,497*	*22,680*
	Stdev	*11,668*	*10,013*
Q7	1	106,910	96,380
	2	43,130	43,210
	3	37,440	36,870
	Average	*62,493*	*58,820*
	Stdev	*38,571*	*32,682*
Q8	1	53,950	57,870
	2	52,290	55,110
	3	2,950	3,410
	Average	*36,397*	*38,797*
	Stdev	*28,978*	*30,677*
Q9	1	507,740	595,430
	2	27,050	27,150
	3	26,930	25,380
	Average	*187,240*	*215,987*
	Stdev	*277,561*	*328,609*
Q10	1	41,830	38,440
	2	13,260	11,900
	3	34,160	34,410
	Average	*29,750*	*28,250*
	Stdev	*14,787*	*14,302*
Q11	1	10,960	11,070
	2	3,230	3,600
	3	537,660	513,410
	Average	*183,950*	*176,027*
	Stdev	*306,346*	*292,206*
Q12	1	30,000	29,950
	2	32,590	34,500
	3	93,710	106,730
	Average	*52,100*	*57,060*
	Stdev	*36,059*	*43,076*
Q13	1	538,320	490,240
	2	31,770	35,520
	3	53,560	52,190
	Average	*207,883*	*192,650*
	Stdev	*286,374*	*257,855*

Query Number	Execution Number	AES-256 (ms)	No-encryption (ms)
Q14	1	27,820	32,840
	2	96,850	88,950
	3	20,470	20,520
	Average	*48,380*	*47,437*
	Stdev	*42,137*	*36,476*
Q15	1	49,580	44,160
	2	50,370	50,400
	3	28,430	28,450
	Average	*42,793*	*41,003*
	Stdev	*12,445*	*11,310*
Q16	1	19,090	17,490
	2	534,730	512,330
	3	29,150	29,940
	Average	*194,323*	*186,587*
	Stdev	*294,844*	*282,171*
Q17	1	9,480	10,620
	2	109,000	101,940
	3	38,950	38,800
	Average	*52,477*	*50,453*
	Stdev	*51,120*	*46,762*
Q18	1	28,730	31,970
	2	20,630	22,870
	3	473,270	466,710
	Average	*174,210*	*173,850*
	Stdev	*259,025*	*253,665*
Q19	1	15,780	13,620
	2	497,060	464,730
	3	57,980	56,700
	Average	*190,273*	*178,350*
	Stdev	*266,522*	*248,946*
Q20	1	58,930	62,240
	2	166,750	35,730
	3	17,090	17,190
	Average	*80,923*	*38,387*
	Stdev	*77,216*	*22,642*
Q21	1	86,810	87,510
	2	8,720	9,990
	3	44,340	44,210
	Average	*46,623*	*47,237*
	Stdev	*39,095*	*38,849*
Q22	1	3,710	3,430

Query Number	Execution Number	AES-256 (ms)	No-encryption (ms)
	2	45,690	46,420
	3	2,990	2,900
	Average	*17,463*	*17,583*
	Stdev	*24,448*	*24,975*
Total	**All Executions**	**6,098,700**	**5,851,760**
Total Stdev	**All Executions**	**145,199**	**142,378**

As expected in this Figure 19 is possible to see AES-256 having overhead in almost all queries, the queries where it shows notable encryption overhead it is, Q2, Q3, Q5, Q7, Q13, Q16, while slightly low the margin of overhead, for this scale factor it is still significant, with an average of 6.38% around this queries when comparing AES-256 to without encryption results. These queries involve subqueries, access to the table Lineitem which is very heavy compared to the other tables, and bi-directional joins. The biggest outlier when looking at Figure 19 is in query 20, taking 2.11 longer to query. Q20 overhead from AES256 encryption can be due to nested subquery execution per row, and access to Lineitem table. These factors combined show that Q20 is particularly sensitive to encryption, at scale factor 1, having an impact on disk I/O.

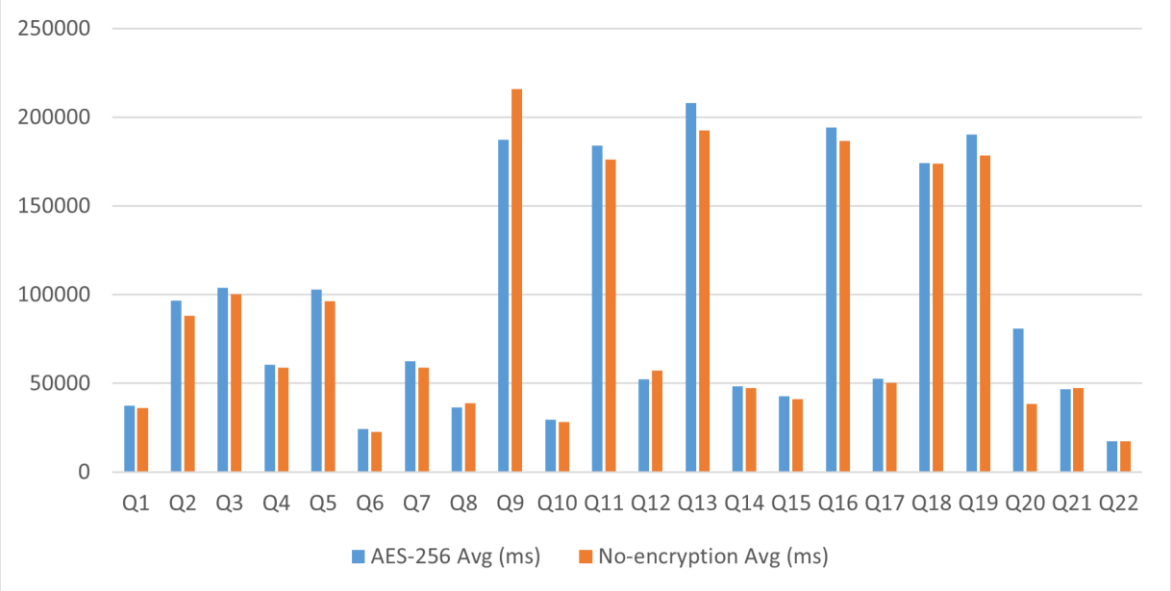


Figure 19 - Average query execution times (in ms) for AES-128, AES-192, AES-256, and No-encryption algorithms across 22 queries in MySQL – Scale Factor 1

5.2.6 MySQL Scale Factor 10

The MySQL executions using a scale factor of 10 were not performed because of the significant time and resources involved in creating the database and executing queries at scale. As the database increases, the time required for both database creation and query execution can grow exponentially, making it impractical to conclude

this study within the time frame. This would introduce challenges in managing system resources and query executions.

5.2.7 MySQL Scale Factor 0.1 V9.2 Enterprise

Table 10 and Figure 20 show the results of the executions in MySQL for scale factor 0.1 for the version 9.2 enterprise. In this section, the performance of the AES encryption algorithm AES-256 is analyzed compared to no-encryption across 22 queries, with each executed three times, while the grand total reflects all 66 executions combined.

In the powertest the average execution time increased from 450.61ms (no-encryption) to 492.73ms (AES-256 encryption) showing an overhead of 9.3% due to encryption computational costs. Additionally, the total standard deviation for AES-256 was 513 ms, compared to 411 ms for no-encryption, showing slightly higher variability in performance under encryption. CPU usage shows that despite the expectation that encryption would take more CPU resources that was not the case, no-encryption had between 45% to 50% and AES-256 encryption had between 20% and 25%. This indicates that when querying with AES-256, more time is spend waiting on the disk operations, resulting in lower CPU utilization. In throughput tests is good to say that exhibits less predictable overheads due to caching effects and I/O scheduling, which may affect results. In Disk time no-encryption ranges from 14% to 45% and AES-256 encryption ranges from 70% to 72%, the time waiting of encrypting/decrypting data on disk operations increases, this is critical in throughput tests, high disk time can limit the overall query throughput, where multiple queries are competing for disk access and that explains why AES-256 took more time to query compared to no-encryption. The increased disk time is consistent with higher variability in AES-256 runs, especially in disk-intensive queries like Q14, where AES-256 had a standard deviation of 250 ms, compared to just 20 ms under no-encryption. In terms of RAM usage, no-encryption has an average of 44% and AES-256 encryption ranges from 31% to 33%, there is a lower RAM usage under AES-256 which can result from system spending more time waiting on disk operations, reducing active memory engagement.

Table 10 - MySQL Scale Factor 0.1 v9.2 enterprise results

Query Number	Execution Number	AES-256 (ms)	No-encryption (ms)
Q1	1	930	930
	2	940	980
	3	900	1,120
	Average	*923*	*1,010*
	Stdev	*21*	*98*
Q2	1	70	30
	2	40	10
	3	10	40

Query Number	Execution Number	AES-256 (ms)	No-encryption (ms)
	Average	*40*	*27*
	Stdev	*30*	*15*
Q3	1	230	250
	2	350	230
	3	270	230
	Average	*283*	*237*
	Stdev	*61*	*12*
Q4	1	100	110
	2	130	130
	3	110	150
	Average	*113*	*130*
	Stdev	*15*	*20*
Q5	1	230	250
	2	240	280
	3	240	380
	Average	*237*	*303*
	Stdev	*6*	*68*
Q6	1	430	260
	2	280	290
	3	250	280
	Average	*320*	*277*
	Stdev	*96*	*15*
Q7	1	200	180
	2	180	180
	3	210	230
	Average	*197*	*197*
	Stdev	*15*	*29*
Q8	1	670	680
	2	640	700
	3	690	680
	Average	*667*	*687*
	Stdev	*25*	*12*
Q9	1	2,880	870
	2	1,060	1,010
	3	990	1,020
	Average	*1,643*	*967*
	Stdev	*1,072*	*84*
Q10	1	260	270
	2	300	330
	3	310	350
	Average	*290*	*317*

Query Number	Execution Number	AES-256 (ms)	No-encryption (ms)
	Stdev	*26*	*42*
Q11	1	30	30
	2	50	30
	3	40	50
	Average	*40*	*37*
	Stdev	*10*	*12*
Q12	1	440	480
	2	480	450
	3	450	450
	Average	*457*	*460*
	Stdev	*21*	*17*
Q13	1	880	760
	2	1,010	750
	3	720	830
	Average	*870*	*780*
	Stdev	*145*	*44*
Q14	1	760	380
	2	820	360
	3	360	400
	Average	*647*	*380*
	Stdev	*250*	*20*
Q15	1	710	840
	2	860	910
	3	720	1,000
	Average	*763*	*917*
	Stdev	*84*	*80*
Q16	1	390	410
	2	230	390
	3	330	290
	Average	*317*	*363*
	Stdev	*81*	*64*
Q17	1	120	50
	2	80	80
	3	40	60
	Average	*80*	*63*
	Stdev	*40*	*15*
Q18	1	1,170	1,190
	2	1,170	1,040
	3	1,030	1,210
	Average	*1,123*	*1,147*
	Stdev	*81*	*93*

Query Number	Execution Number	AES-256 (ms)	No-encryption (ms)
Q19	1	30	40
	2	40	40
	3	20	40
	Average	*30*	*40*
	Stdev	*10*	*0*
Q20	1	340	60
	2	180	170
	3	140	180
	Average	*220*	*137*
	Stdev	*106*	*67*
Q21	1	1,700	1,530
	2	1,590	1,390
	3	1,340	1,330
	Average	*1,543*	*1,417*
	Stdev	*184*	*103*
Q22	1	20	20
	2	20	20
	3	20	30
	Average	*20*	*23*
	Stdev	*0*	*6*
Total	**All Executions**	**32,470**	**29,740**
Total Stdev	**All Executions**	**513**	**411**

Encryption typically adds overhead, analysis of the average query times shows that AES-256 encryption does not generally degrade performance. Queries such as Q9 and Q14 experience significant overhead – indicating high usage of resources indicating heightened I/O or CPU demands due to encryption. Queries such as Q5 and Q15 had faster execution times with encryption, which shows that encryption does not strictly introduce overhead in all queries.

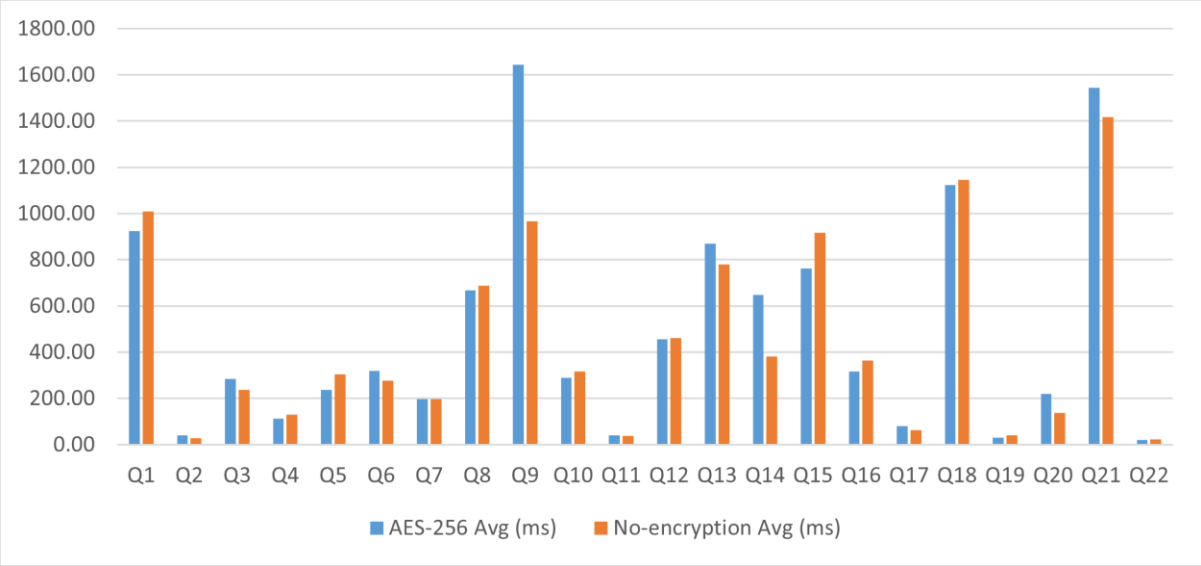


Figure 20 - Average query execution times (in ms) for AES-256, and No-encryption algorithms across 22 queries in MySQL – Scale Factor 0.1 v9.2 Enterprise

5.2.8 MySQL Scale Factor 1 V9.2 Enterprise

Table 11 and Figure 21 show the results of the executions in MySQL for scale factor 1 for the version 9.2 enterprise. In this section, the performance of the AES encryption algorithm AES-256 is analyzed compared to no-encryption across 22 queries, with each executed three times. AES-256 encryption shows an overhead increase of 1% over no-encryption execution time when comparing the first execution of each query. However, in concurrent executions, throughput test, AES-256, have an increase of time ranging between 30% to 40% in total execution time when comparing individually each set of executions inside of throughput. The variability in execution times is especially visible in queries like Q1, where AES-256 has a standard deviation of 19,189 ms, compared to just 886 ms for no-encryption. Similarly, Q13 shows high deviations under both configurations, with 35,757 ms for AES-256 and 46,187 ms for no-encryption, pointing to inconsistent performance on large joins. Something important to notice is that in powertest, queries Q3, Q9, and Q13 are sometimes faster with AES-256, but in throughput test the same queries are slower with AES-256. This contrast can come from changed execution plans or caching behavior. However, the added I/O and CPU overhead of AES-256 impacts the performance under heavier query concurrency. Each test shows average usage of RAM, CPU, and Disk time. In the powertest, AES-256 showed 7.56% higher CPU, and 15.2% higher Disk Time compared to no-encryption. RAM is lower with AES-256, which can happen if the time waiting for encrypting/decrypting data on disk operations increases, this is critical in throughput tests, having less time actively using memory. On Throughput, there is a higher consume of RAM, and a higher usage of Disk Time, where no-encryption with 68.37% and AES-256 with 71.97%, suggesting that when querying encrypted data, the database spends more time in I/O operations. Throughput has multiple concurrent queries, which can add more overhead when

passing through AES-256, slowing down disk operations, which is also reflected in queries such as Q20 and Q21, where the standard deviation under AES-256 was 3,952 ms and 3,909 ms, compared to 7,911 ms and 9,191 ms for no-encryption, causing large increase in total runtime. The difference between powertest and throughput test shows why it's important to benchmark under realistic scenarios to see the true cost of encryption.

Table 11 - MySQL Scale Factor 1 v9.2 enterprise results

Query Number	**Execution Number**	**AES-256 (ms)**	**No-encryption (ms)**
Q1	1	46,600	19,230
	2	13,760	18,700
	3	12,980	20,430
	Average	*24,447*	*19,453*
	Stdev	*19,189*	*886*
Q2	1	1,010	1,200
	2	660	720
	3	1,150	930
	Average	*940*	*950*
	Stdev	*252*	*241*
Q3	1	138,940	154,120
	2	177,920	113,350
	3	153,450	112,470
	Average	*156,770*	*126,647*
	Stdev	*19,701*	*23,797*
Q4	1	22,080	19,730
	2	19,160	15,100
	3	17,290	16,180
	Average	*19,510*	*17,003*
	Stdev	*2,414*	*2,422*
Q5	1	27,260	21,340
	2	17,850	14,400
	3	21,480	16,450
	Average	*22,197*	*17,397*
	Stdev	*4,746*	*3,566*
Q6	1	12,220	9,950
	2	6,780	7,560
	3	7,040	7,200
	Average	*8,680*	*8,237*
	Stdev	*3,068*	*1,495*
Q7	1	73,770	62,360
	2	60,280	39,180

Query Number	Execution Number	AES-256 (ms)	No-encryption (ms)
	3	64,160	38,040
	Average	*66,070*	*46,527*
	Stdev	*6,945*	*13,724*
Q8	1	34,690	36,180
	2	33,600	27,840
	3	35,860	54,310
	Average	*34,717*	*39,443*
	Stdev	*1,130*	*13,533*
Q9	1	238,950	268,740
	2	250,500	200,110
	3	265,240	172,140
	Average	*251,563*	*213,663*
	Stdev	*13,177*	*49,706*
Q10	1	29,600	21,970
	2	22,540	19,680
	3	27,700	15,700
	Average	*26,613*	*19,117*
	Stdev	*3,653*	*3,173*
Q11	1	21,000	14,570
	2	14,560	12,780
	3	16,840	11,380
	Average	*17,467*	*12,910*
	Stdev	*3,265*	*1,599*
Q12	1	21,810	15,190
	2	11,690	12,310
	3	10,860	13,400
	Average	*14,787*	*13,633*
	Stdev	*6,097*	*1,454*
Q13	1	245,360	279,100
	2	313,170	220,310
	3	298,940	188,000
	Average	*285,823*	*229,137*
	Stdev	*35,757*	*46,187*
Q14	1	26,650	26,520
	2	18,190	13,910
	3	21,010	20,860
	Average	*21,950*	*20,430*
	Stdev	*4,308*	*6,316*
Q15	1	42,090	21,830
	2	12,460	19,970
	3	14,560	20,870

Query Number	Execution Number	AES-256 (ms)	No-encryption (ms)
	Average	*23,037*	*20,890*
	Stdev	*16,534*	*930*
Q16	1	11,540	11,130
	2	11,230	8,270
	3	9,760	8,970
	Average	*10,843*	*9,457*
	Stdev	*951*	*1,491*
Q17	1	3,930	3,810
	2	4,130	3,480
	3	2,980	2,320
	Average	*3,680*	*3,203*
	Stdev	*614*	*783*
Q18	1	15,810	15,080
	2	12,830	11,210
	3	14,700	10,790
	Average	*14,447*	*12,360*
	Stdev	*1,506*	*2,365*
Q19	1	9,930	7,090
	2	5,630	5,250
	3	6,920	3,530
	Average	*7,493*	*5,290*
	Stdev	*2,207*	*1,780*
Q20	1	30,770	34,110
	2	36,780	22,400
	3	38,220	19,040
	Average	*35,257*	*25,183*
	Stdev	*3,952*	*7,911*
Q21	1	47,360	49,550
	2	43,010	33,200
	3	50,810	34,100
	Average	*47,060*	*38,950*
	Stdev	*3,909*	*9,191*
Q22	1	2,130	1,650
	2	1,270	1,650
	3	1,490	1,420
	Average	*1,630*	*1,573*
	Stdev	*447*	*133*
Total	**All Executions**	**3,284,940**	**2,704,360**
Total Stdev	**All Executions**	**76,891**	**63,916**

When examining the average or AES-256 and no-encryption in scale factor 1 using enterprise version in version 9.2 is possible to see that queries that process large data sets such as Q3, Q9, and Q13 show significantly slower execution times, showing that encryption takes I/O and CPU demands. Although many queries show similar performance between AES-256 encryption and no-encryption, AES-256 adds more overhead compared to no-encryption.

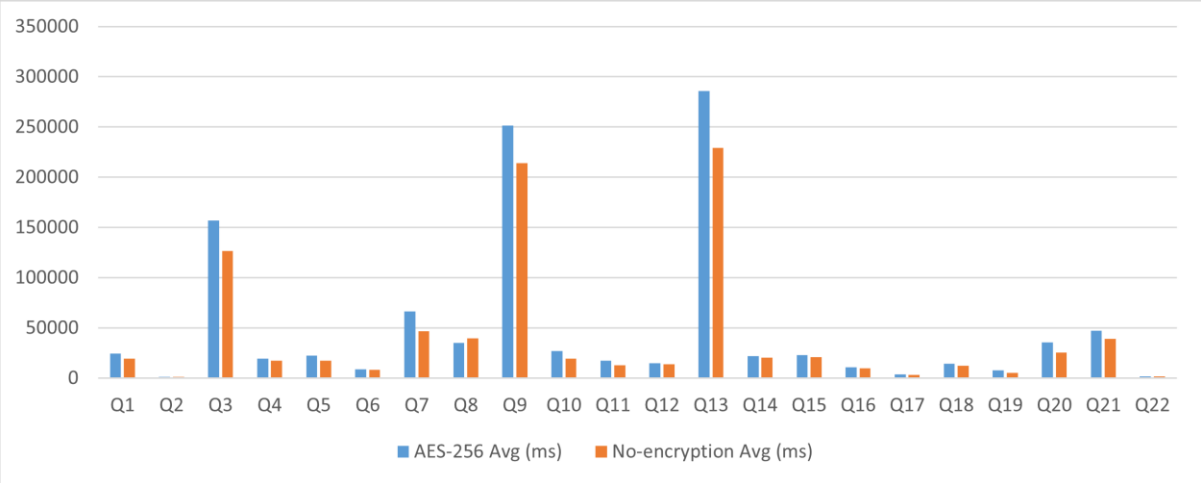


Figure 21 - Average query execution times (in ms) for AES-256, and No-encryption algorithms across 22 queries in MySQL – Scale Factor 1 v9.2 Enterprise

5.2.9 SQL Server Scale factor 0.1

Table 12 and Figure 22 show the results for the executions in SQL Server for the scale factor 0.1. This section analysed the performance of the AES encryption algorithms AES-128, AES-192, AES-256 compared to the non-encryption across 22 queries executed three times, while the grand total reflects all 66 executions combined.

The results tell us that AES encryption algorithms can sometimes outperform non-encrypted processes in some cases such as Q16. The results tell us that AES-192 generally performs better than AES-128 and AES-256, with lower execution times. For example, AES-128 ranges from 62ms to 2,592ms with a standard deviation of 511 ms, while AES-256 ranges from 43ms to 2,375ms with a standard deviation of 402 ms. AES-192 shows a better performance, ranging from 51ms to 2,330ms, with the lowest standard deviation of 361 ms among the encryption methods, which can be explained by the scale factor number 0.1 which can give more variations in terms of execution times and system performance measurements and encryption operations can benefit from better memory/cache consumption, giving faster results. The no-encryption baseline ranges from 74ms to 2,525ms, indicating that encryption does not always introduce a significant overhead.

The no-encryption configuration processing times show greater inconsistency because unencrypted data handling is less efficient especially in I/O operations or memory usage The small dataset size around 100 MB could have increased these

fluctuations. AES-256 performs better than no-encryption in queries such as Q16 which indicates that encryption can offer performance benefits even in smaller datasets. This may be influenced by the AES-NI support (Appendix D), reducing the CPU overhead, making the encryption and decryption processes when querying as efficient as a non-encryption execution or even more efficient.

AES-192 is the most efficient encryption algorithm for small datasets, with lower computational overhead, followed by AES-256 and AES-128. From the total, it is possible to see the difference between the algorithms which AES-128 takes more time to query than the rest of the algorithms and it's possible to see how AES-192 outperforms all the algorithms, having a total query time of 16,226ms compared to AES-128 21,689ms and AES-256 19,085ms.

Table 12 – SQL Server Scale Factor 0.1 results

Query Number	**Execution Number**	**AES-128 (ms)**	**AES-192 (ms)**	**AES-256 (ms)**	**No-encryption (ms)**
Q1	1	231	235	121	229
	2	263	204	321	289
	3	335	179	335	406
	Average	*276*	*206*	*259*	*308*
	Stdev	*43*	*23*	*98*	*74*
Q2	1	62	81	174	82
	2	253	440	270	239
	3	123	56	124	122
	Average	*146*	*192*	*189*	*148*
	Stdev	*80*	*175*	*61*	*67*
Q3	1	183	167	111	261
	2	101	107	125	120
	3	164	106	293	215
	Average	*149*	*127*	*176*	*199*
	Stdev	*35*	*29*	*83*	*59*
Q4	1	133	145	77	239
	2	139	102	122	185
	3	156	86	136	164
	Average	*143*	*111*	*112*	*196*
	Stdev	*10*	*25*	*25*	*32*
Q5	1	167	203	112	218
	2	92	93	135	160
	3	132	90	120	147
	Average	*130*	*129*	*122*	*175*
	Stdev	*31*	*53*	*10*	*31*
Q6	1	71	69	76	74
	2	65	57	78	73

Query Number	**Execution Number**	**AES-128 (ms)**	**AES-192 (ms)**	**AES-256 (ms)**	**No-encryp-tion (ms)**
	3	71	51	53	94
	Average	*69*	*59*	*69*	*80*
	Stdev	*3*	*7*	*11*	*10*
Q7	1	101	103	70	149
	2	88	84	138	132
	3	125	100	103	128
	Average	*105*	*96*	*104*	*136*
	Stdev	*15*	*8*	*28*	*9*
Q8	1	228	95	199	278
	2	171	386	183	172
	3	213	113	132	166
	Average	*204*	*198*	*171*	*205*
	Stdev	*24*	*133*	*29*	*51*
Q9	1	274	350	447	280
	2	199	148	230	228
	3	223	181	243	226
	Average	*232*	*226*	*307*	*245*
	Stdev	*31*	*88*	*99*	*25*
Q10	1	83	78	43	89
	2	98	138	95	93
	3	80	56	93	91
	Average	*87*	*91*	*77*	*91*
	Stdev	*8*	*35*	*24*	*2*
Q11	1	282	286	135	364
	2	216	149	322	335
	3	278	121	278	303
	Average	*259*	*185*	*245*	*334*
	Stdev	*30*	*72*	*80*	*25*
Q12	1	127	113	80	125
	2	144	89	210	146
	3	266	142	188	224
	Average	*179*	*115*	*159*	*165*
	Stdev	*62*	*22*	*57*	*43*
Q13	1	2,521	1,607	1,239	2,525
	2	2,592	2,330	2,112	2,450
	3	2,525	1,196	2,375	2,488
	Average	*2,546*	*1,711*	*1,909*	*2,488*
	Stdev	*33*	*469*	*486*	*31*
Q14	1	209	168	805	145
	2	155	164	156	134

Query Number	**Execution Number**	**AES-128 (ms)**	**AES-192 (ms)**	**AES-256 (ms)**	**No-encryp-tion (ms)**
	3	137	88	129	181
	Average	*167*	*140*	*363*	*153*
	Stdev	*31*	*37*	*313*	*20*
Q15	1	84	102	44	95
	2	105	64	57	89
	3	148	92	148	142
	Average	*112*	*86*	*83*	*109*
	Stdev	*27*	*16*	*46*	*24*
Q16	1	674	639	301	737
	2	533	325	409	571
	3	525	264	429	541
	Average	*577*	*409*	*380*	*616*
	Stdev	*68*	*164*	*56*	*86*
Q17	1	192	158	122	186
	2	196	105	197	210
	3	211	110	150	232
	Average	*200*	*124*	*156*	*209*
	Stdev	*8*	*24*	*31*	*19*
Q18	1	918	640	578	1,094
	2	570	388	713	744
	3	734	331	720	719
	Average	*741*	*453*	*670*	*852*
	Stdev	*142*	*134*	*65*	*171*
Q19	1	97	136	77	124
	2	63	61	76	78
	3	68	53	82	79
	Average	*76*	*83*	*78*	*94*
	Stdev	*15*	*37*	*3*	*21*
Q20	1	451	388	327	446
	2	429	206	409	422
	3	352	200	428	440
	Average	*411*	*265*	*388*	*436*
	Stdev	*42*	*87*	*44*	*10*
Q21	1	197	95	183	309
	2	250	598	214	306
	3	223	156	326	327
	Average	*223*	*283*	*241*	*314*
	Stdev	*22*	*224*	*61*	*9*
Q22	1	181	173	81	196
	2	252	108	253	267

Query Number	Execution Number	AES-128 (ms)	AES-192 (ms)	AES-256 (ms)	No-encryption (ms)
	3	160	78	145	170
	Average	*198*	*120*	*160*	*211*
	Stdev	*39*	*40*	*71*	*41*
Total	**All Executions**	**21,689**	**16,226**	**19,257**	**23,293**
Total Stdev	**All Executions**	**511**	**361**	**402**	**502**

In Figure 22, it is possible to see that in most cases no-encryption has a bad performance compared to the encryption algorithms. This might not be expected but can be due to performance variance at low scale factors, where AES-192 had the fastest average execution 5,409ms while non-encryption had 7,764ms being the slowest which might alter how the execution time runs might appear, one other cause is the I/O behavior where encryption overhead is none or beneficial depending on how the system deals with encryption processing query. But still follows most of the encryption algorithms execution times. AES-128 had the worst performance among the encryption algorithms while AES-192 and AES-256 were relatively close to each other which is possible to say that, while AES-192 outperforms all the algorithms, to have a better security for a smaller dataset its advised to use AES-256 which is not going to have a significant effect on the performance when compared to AES-192.

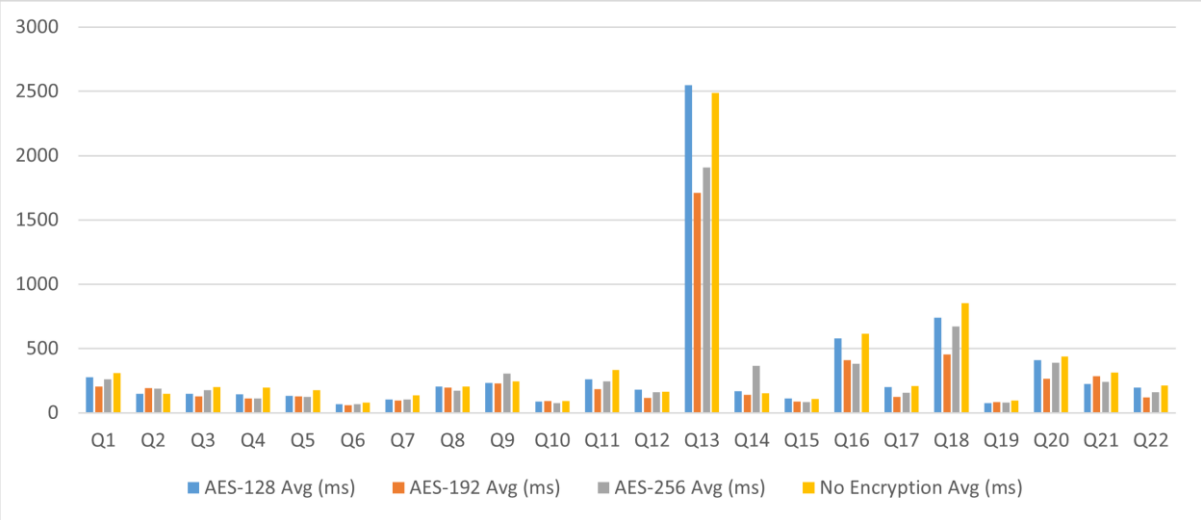


Figure 22 - Average query execution times (in ms) for AES-128, AES-192, AES-256, and No-encryption algorithms across 22 queries in SQL Server – factor 0.1

5.2.10 SQL Server Scale Factor 1

Table 13 and Figure 23 show the results for the executions in SQL Server for the scale factor 1. In this section, the performance of the AES encryption algorithms (AES-128, AES-192, and AES-256) is analysed and compared to no-encryption across 22 queries, with each query executed three times, while the grand total reflects all 66 executions combined.

The analysis shows that AES-128 performs more efficiently than AES-192 and AES-256, with a range from 107ms to 8,196ms, while AES-256 ranges from 124ms to 7,170ms, which has a common range with no-encryption which ranges from 99ms to 7,986ms. AES-128 also demonstrated the lowest total standard deviation among the encryption algorithms, with 1,428 ms, indicating more consistent performance. In contrast, AES-192 had the highest variability at 1,583 ms, and AES-256 had 1,560 ms. No-encryption reached 1,565 ms in standard deviation. AES-192 shows a more moderate performance with an execution time range from 94ms to 8,495ms. No-encryption exhibits inconsistencies in processing times, as seen in queries like Q9 on execution one, where for the AES-256, the execution time was 2,716ms and for no-encryption, 822ms. The standard deviation for AES-256 in Q9 was 902 ms, significantly higher than the 38 ms observed under no-encryption, reinforcing the inconsistency seen in encrypted runs. These results are likely caused by I/O operations or memory management. AES-128 can efficiently be employed with manageable impact on system performance, where it shows a most stable performance with lower execution times, whereas AES-256 and AES-192 introduce additional overhead due to their key sizes.

There were some outliers, such as Q18, where execution times are over 8,000ms for some algorithms. This suggests that some test cases might require more complex data processing tasks, leading to slower performance for encryption and no-encryption algorithms. Q18 is designed to find high-value customers who have purchased a considerable quantity of items, and this query accesses the Orders table, Customer table and Lineitem, the largest table among TPC-H generated tables. Q18 outliers observed come from an existent sub-query and from joins across large tables. The subquery requires SQL Server in each line of the Lineitem table to do SUM(L_QUANTITY), which can be resource-intensive, especially on large datasets. To improve the performance, views and indexed views can be created, which is going to reduce the need to repeatedly calculate the SUM(L_QUANTITY), allowing the query to fetch aggregated data directly from the view and reducing the computation complexity when executing this query, especially when encryption is involved. In terms of total variability, AES-128 again performed best with the lowest overall standard deviation, while AES-192 and AES-256 had the highest. From the total, it is possible to see the difference between the algorithms, in which AES-256 takes more time to query than the rest. It is possible to see how AES-128 outperforms all the algorithms, having a total query time of 54,454ms compared to AES-192 57,879ms and AES-256 61,178ms. This shows in a higher scale factor of 1gb the

correlation between the algorithms being expected to have this behaviour. No-encryption total execution time performance was worse than AES-128, which can be explained by different factors, such as AES-NI hardware acceleration that can contribute to better CPU and I/O resources or memory management.

Table 13 – SQL Server Scale Factor 1 results

Query Number	**Execution Number**	**AES-128 (ms)**	**AES-192 (ms)**	**AES-256 (ms)**	**No-encryption (ms)**
Q1	1	316	313	316	316
	2	318	316	367	316
	3	605	630	726	670
	Average	*413*	*420*	*470*	*434*
	Stdev	*136*	*149*	*182*	*167*
Q2	1	169	167	829	169
	2	161	240	136	138
	3	123	118	124	119
	Average	*151*	*175*	*363*	*142*
	Stdev	*20*	*50*	*330*	*21*
Q3	1	740	333	522	299
	2	357	414	418	341
	3	378	368	376	376
	Average	*492*	*372*	*439*	*339*
	Stdev	*176*	*33*	*61*	*31*
Q4	1	853	844	563	856
	2	552	559	565	559
	3	584	603	684	588
	Average	*663*	*669*	*604*	*668*
	Stdev	*135*	*125*	*57*	*134*
Q5	1	225	236	218	210
	2	510	513	453	493
	3	228	251	234	265
	Average	*321*	*333*	*302*	*323*
	Stdev	*134*	*127*	*107*	*123*
Q6	1	248	253	280	265
	2	352	372	325	392
	3	281	287	294	291
	Average	*294*	*304*	*300*	*316*
	Stdev	*43*	*50*	*19*	*55*
Q7	1	297	303	289	301
	2	523	541	547	601
	3	323	291	292	307
	Average	*381*	*378*	*376*	*403*
	Stdev	*101*	*115*	*121*	*140*

Query Number	**Execution Number**	**AES-128 (ms)**	**AES-192 (ms)**	**AES-256 (ms)**	**No-encryp-tion (ms)**
Q8	1	665	630	918	711
	2	317	503	319	307
	3	295	294	312	346
	Average	*426*	*476*	*516*	*455*
	Stdev	*169*	*139*	*284*	*182*
Q9	1	727	748	2,716	822
	2	753	739	796	751
	3	724	703	811	734
	Average	*735*	*730*	*1,441*	*769*
	Stdev	*13*	*19*	*902*	*38*
Q10	1	295	278	287	294
	2	314	613	292	309
	3	297	333	336	294
	Average	*302*	*408*	*305*	*299*
	Stdev	*9*	*147*	*22*	*7*
Q11	1	153	162	169	154
	2	441	459	413	454
	3	107	94	138	99
	Average	*234*	*238*	*240*	*236*
	Stdev	*148*	*158*	*123*	*156*
Q12	1	541	567	529	521
	2	547	528	555	531
	3	536	518	561	539
	Average	*541*	*538*	*548*	*530*
	Stdev	*5*	*21*	*14*	*7*
Q13	1	3,487	3,384	3,345	3,334
	2	3,366	3,714	3,260	3,332
	3	3,416	3,383	3,309	3,616
	Average	*3,423*	*3,494*	*3,305*	*3,427*
	Stdev	*50*	*156*	*35*	*133*
Q14	1	891	584	2,144	754
	2	422	579	429	433
	3	423	429	431	421
	Average	*579*	*531*	*1,001*	*536*
	Stdev	*221*	*72*	*808*	*154*
Q15	1	229	242	254	234
	2	246	250	242	237
	3	237	231	255	243
	Average	*237*	*241*	*250*	*238*
	Stdev	*7*	*8*	*6*	*4*

Query Number	**Execution Number**	**AES-128 (ms)**	**AES-192 (ms)**	**AES-256 (ms)**	**No-encryption (ms)**
Q16	1	566	577	548	537
	2	595	540	542	534
	3	586	591	576	631
	Average	*582*	*569*	*555*	*567*
	Stdev	*12*	*22*	*15*	*45*
Q17	1	227	219	257	238
	2	241	286	261	243
	3	221	224	221	221
	Average	*230*	*243*	*246*	*234*
	Stdev	*8*	*30*	*18*	*9*
Q18	1	8,196	8,495	8,198	7,986
	2	6,591	6,908	6,287	6,633
	3	4,489	6,755	7,170	7,381
	Average	*6,425*	*7,386*	*7,218*	*7,333*
	Stdev	*1,518*	*787*	*781*	*553*
Q19	1	286	294	292	283
	2	252	255	264	253
	3	258	270	252	258
	Average	*265*	*273*	*269*	*265*
	Stdev	*15*	*16*	*17*	*13*
Q20	1	293	286	293	333
	2	298	281	273	237
	3	241	252	254	261
	Average	*277*	*273*	*273*	*277*
	Stdev	*26*	*15*	*16*	*41*
Q21	1	733	735	1,018	1,066
	2	1,374	1,542	1,592	1,194
	3	751	757	755	797
	Average	*953*	*1,011*	*1,122*	*1,019*
	Stdev	*298*	*375*	*349*	*165*
Q22	1	254	271	319	229
	2	222	217	218	217
	3	208	207	209	230
	Average	*228*	*232*	*249*	*225*
	Stdev	*19*	*28*	*50*	*6*
Total	**All Executions**	**54,454**	**57,879**	**61,178**	**57,104**
Total Stdev	**All Executions**	**1,428**	**1,583**	**1,560**	**1,565**

On Figure 23, it is possible to see AES-256 outliers, in specific in Q2, Q9, Q14 and Q21. Q9 involves a subquery and complex joins with Lineitem which makes the query slower when querying a database encrypted with AES-256. Q14 doesn't involve subquery but accesses to Lineitem table which is the biggest table among the other tables which affects the query times when this table is involved. On Q21 the difference with the other encryption algorithms was not so significant but as Q14 had access to Lineitem table which says that when querying encrypted databases its possible to see in most cases significant overhead when complex joins and access to large tables which sometimes full scans are needed.

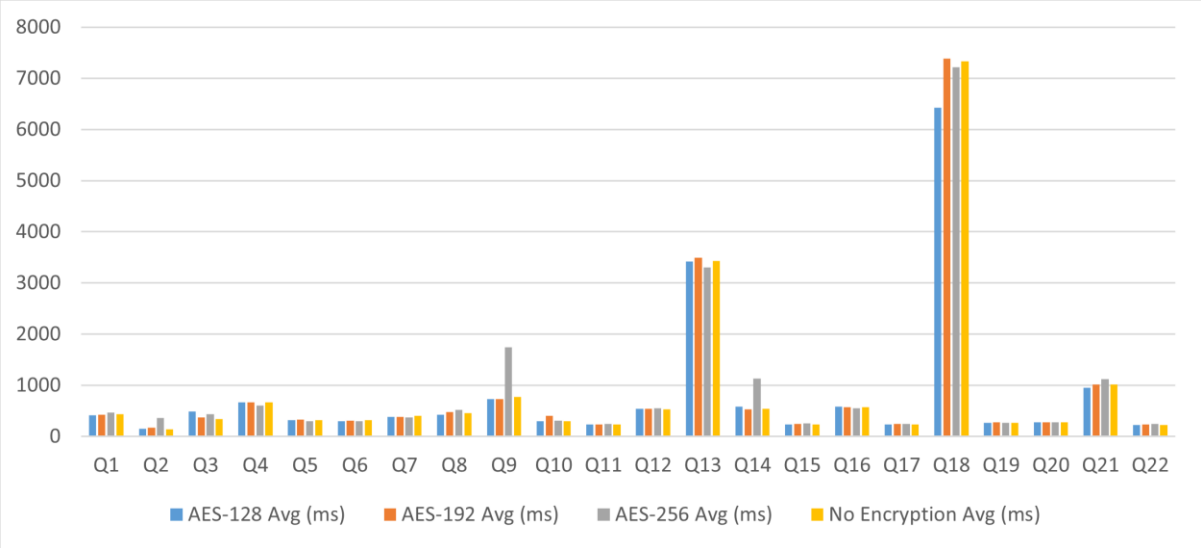


Figure 23 - Average query execution times (in ms) for AES-128, AES-192, AES-256, and No-encryption algorithms across 22 queries in SQL Server – factor 1

5.2.11 SQL Server Scale Factor 10

The following Table 14 and Figure 24 show the results for the executions in MySQL for the scale factor 10. In this section, the performance of the AES encryption algorithms (AES-128, AES-192, and AES-256) is compared to no-encryption across 22 queries, with each query executed three times, while the grand total reflects all 66 executions combined.

AES-128 performs efficiently, with execution times ranging from 2,017ms to 35,249ms. It also has a total standard deviation of 7,271 ms, indicating moderate variability across runs. AES-256, while having a key size of 256 bits, displays times ranging from 2,641 ms to 36,016 ms, and shows the highest total standard deviation of 8,450 ms, highlighting inconsistency between test runs. This variability is close to no-encryption, which ranged from 963 ms to 35,612 ms and had a total standard deviation of 6,882 ms. AES-192 shows moderate performance, with times between 981ms and 33,696ms.

AES-192 shows the most stable and efficient performance among the AES test runs, with the lowest total standard deviation of 6,921 ms, making it the most effective

encryption algorithm for scale factor 10. While having additional overhead at AES-192 and AES-256, AES-192 maintains a good balance between performance and consistency, where AES-256 is more impacted and significantly slower. The difference is minimal, but AES-192 outperformed AES-128 and AES-256 in terms of total execution times. At a scale factor of 10, it is already possible to see a more stable total execution time of the no-encryption, which outperformed all the key sizes of the AES encryption algorithm.

Table 14 - SQL Server Scale Factor 10 results

Query Number	**Execution Number**	**AES-128 (ms)**	**AES-192 (ms)**	**AES-256 (ms)**	**No-encryption (ms)**
Q1	1	4,628	4,561	4,706	4,456
	2	4,830	4,546	4,516	4,453
	3	4,395	4,682	4,315	4,631
	Average	*4,618*	*4,596*	*4,512*	*4,513*
	Stdev	*178*	*61*	*160*	*83*
Q2	1	2,017	981	2,454	3,225
	2	2,513	2,539	2,935	2,269
	3	2,462	2,477	2,548	802
	Average	*2,331*	*1,999*	*2,646*	*2,099*
	Stdev	*223*	*720*	*208*	*996*
Q3	1	5,838	6,947	9,314	5,045
	2	5,243	5,080	7,254	4,710
	3	6,673	7,063	6,538	5,413
	Average	*5,918*	*6,363*	*7,702*	*5,056*
	Stdev	*587*	*909*	*1,177*	*287*
Q4	1	7,702	8,896	8,423	6,099
	2	4,162	4,576	4,495	4,214
	3	4,646	4,562	4,309	4,352
	Average	*5,503*	*6,011*	*5,742*	*4,888*
	Stdev	*1,567*	*2,040*	*1,897*	*858*
Q5	1	6,438	7,188	6,549	4,702
	2	5,884	5,004	4,993	5,371
	3	5,250	6,174	6,066	4,516
	Average	*5,857*	*6,122*	*5,869*	*4,863*
	Stdev	*485*	*892*	*650*	*367*
Q6	1	3,755	3,953	4,826	4,037
	2	3,891	3,504	3,675	4,642
	3	3,400	3,651	3,753	3,710
	Average	*3,682*	*3,703*	*4,085*	*4,130*
	Stdev	*207*	*187*	*525*	*386*
Q7	1	4,473	5,306	5,325	4,263
	2	6,472	8,437	8,517	5,261

Query Number	**Execution Number**	**AES-128 (ms)**	**AES-192 (ms)**	**AES-256 (ms)**	**No-encryption (ms)**
	3	4,436	5,180	4,654	4,407
	Average	*5,127*	*6,308*	*6,165*	*4,644*
	Stdev	*951*	*1,507*	*1,685*	*440*
Q8	1	6,809	5,633	7,075	4,837
	2	8,220	8,035	8,756	6,825
	3	7,025	7,377	6,967	6,432
	Average	*7,351*	*7,015*	*7,599*	*6,031*
	Stdev	*621*	*1,013*	*819*	*860*
Q9	1	12,576	12,001	17,723	12,223
	2	13,297	13,192	14,181	11,028
	3	14,262	14,989	14,349	12,298
	Average	*13,378*	*13,394*	*15,418*	*11,850*
	Stdev	*691*	*1,228*	*1,632*	*582*
Q10	1	7,770	5,670	9,614	5,747
	2	5,435	5,080	7,116	5,389
	3	6,876	7,522	7,934	5,989
	Average	*6,694*	*6,091*	*8,221*	*5,708*
	Stdev	*962*	*1,040*	*1,040*	*246*
Q11	1	2,616	2,751	2,641	963
	2	2,371	2,656	3,063	2,225
	3	2,503	2,289	2,767	1,731
	Average	*2,497*	*2,565*	*2,824*	*1,640*
	Stdev	*100*	*199*	*177*	*519*
Q12	1	7,876	7,775	8,460	7,349
	2	9,083	9,321	9,022	9,048
	3	7,577	7,861	7,408	7,485
	Average	*8,179*	*8,319*	*8,297*	*7,961*
	Stdev	*651*	*709*	*669*	*771*
Q13	1	35,249	34,403	36,016	33,192
	2	35,638	35,592	35,093	35,612
	3	35,031	33,696	33,393	32,927
	Average	*35,306*	*34,564*	*34,834*	*33,910*
	Stdev	*251*	*782*	*1,086*	*1,208*
Q14	1	8,060	11,478	19,874	16,943
	2	5,854	5,659	5,685	5,272
	3	5,593	5,998	5,192	4,982
	Average	*6,502*	*7,712*	*10,250*	*9,066*
	Stdev	*1,107*	*2,667*	*6,808*	*5,571*
Q15	1	9,022	7,710	7,406	3,552
	2	4,956	4,268	4,466	3,927

Query Number	**Execution Number**	**AES-128 (ms)**	**AES-192 (ms)**	**AES-256 (ms)**	**No-encryption (ms)**
	3	3,824	5,032	4,035	3,379
	Average	*5,934*	*5,670*	*5,302*	*3,619*
	Stdev	*2,232*	*1,476*	*1,498*	*229*
Q16	1	4,204	4,924	3,580	3,183
	2	3,907	3,885	3,587	3,002
	3	3,357	3,517	3,531	2,892
	Average	*3,823*	*4,109*	*3,566*	*3,026*
	Stdev	*351*	*596*	*25*	*120*
Q17	1	6,978	6,831	8,991	6,802
	2	6,969	6,631	6,660	6,788
	3	7,309	7,393	7,513	6,976
	Average	*7,085*	*6,952*	*7,721*	*6,855*
	Stdev	*158*	*323*	*963*	*86*
Q18	1	22,168	14,878	29,430	13,999
	2	19,102	17,804	31,336	17,172
	3	20,117	19,146	30,497	16,683
	Average	*20,462*	*17,276*	*30,421*	*15,951*
	Stdev	*1,275*	*1,782*	*780*	*1,395*
Q19	1	4,253	4,794	5,113	4,358
	2	3,969	4,344	3,892	3,909
	3	4,392	4,149	4,190	3,959
	Average	*4,205*	*4,429*	*4,398*	*4,075*
	Stdev	*176*	*270*	*520*	*201*
Q20	1	8,007	7,103	8,271	5,040
	2	6,501	6,215	6,290	4,889
	3	5,726	5,946	5,778	4,830
	Average	*6,745*	*6,421*	*6,780*	*4,920*
	Stdev	*947*	*494*	*1,075*	*88*
Q21	1	14,012	13,207	16,991	11,890
	2	12,962	12,161	19,719	12,554
	3	13,635	14,549	14,762	11,255
	Average	*13,536*	*13,306*	*17,157*	*11,900*
	Stdev	*434*	*977*	*2,027*	*530*
Q22	1	2,758	2,822	2,991	2,516
	2	2,933	3,048	2,753	2,978
	3	2,376	2,565	2,601	2,472
	Average	*2,689*	*2,812*	*2,782*	*2,655*
	Stdev	*233*	*197*	*161*	*229*
Total	**All Executions**	**532,266**	**527,207**	**606,877**	**478,080**

Query Number	Execution Number	AES-128 (ms)	AES-192 (ms)	AES-256 (ms)	No-encryption (ms)
Total Stdev	All Executions	7,271	6,921	8,450	6,882

On Figure 24 is possible to see that AES-256 has the highest overhead among the three encryption AES key sizes, and AES-128 performs better than AES-192 and AES-256. Compared to no-encryption results there is some significant overhead among some queries, from the Figure 24 is possible to see some of them, being Q3, Q9, Q18, and Q21, with overhead of 52.33%, 30.11%, 90.71%, and 44.18% respectively. These queries are particularly I/O intensive, involving large tables like Lineitem, and subqueries or aggregations, which can be more sensitive to encryption overhead as shown in the Figure 24. From this Figure 24, the rest of the queries show minimal overhead, but among all the encryption AES key sizes, AES-128 may be preferred.

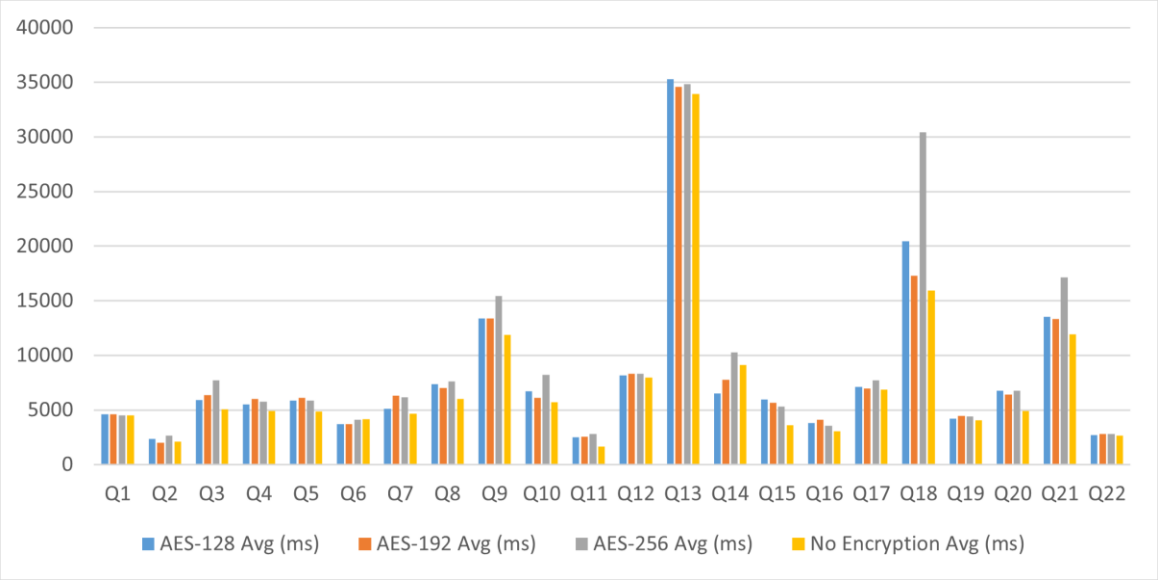


Figure 24 - Average query execution times (in ms) for AES-128, AES-192, AES-256, and No-encryption algorithms across 22 queries in SQL Server – Scale Factor 10

6 DISCUSSION OF THE RESULTS

This chapter summarizes the findings from the performance tests conducted. It compares the algorithms based on the execution time, CPU, RAM, and scalability to provide a comprehensive understanding of the impact of encryption algorithms on databases. In Table 15, it is possible to see the average of the results per scale factor, database, and encryption algorithm.

Table 15 – Average Execution Time, RAM, CPU, and Disk Usage by Encryption Algorithm and Scale Factor

SF	Database	Encryption algorithm	Execution Average (ms)	StdDev Average (ms)	RAM (%)	CPU (%)	Disk Time (%)
0.1	Oracle	AES-128	9,203.33	5,221	36.78	39.16	0.73
0.1	Oracle	AES-192	8,903.33	5,790	27.25	51.66	27.91
0.1	Oracle	AES-256	9,016.67	4,429	28.25	21.06	2.00
0.1	Oracle	no-encryption	8,366.67	3,045	48.28	34.75	0.64
1	Oracle	AES-128	994,026.67	7,056	53.06	32.21	61.34
1	Oracle	AES-192	942,346.67	48,280	50.2	19.06	127.54
1	Oracle	AES-256	754,733.33	472,856	45.59	27.44	31.02
1	Oracle	no encryption	210,963.33	71,074	46.52	31.38	73.19
10	Oracle	AES-128	1,540,676.67	12,470	55.45	37.61	60.02
10	Oracle	AES-192	1,876,790.00	176,137	46.62	23.46	63.37
10	Oracle	AES-256	8,124,546.67	2,990,971	60.02	29.66	42.40
10	Oracle	no encryption	1,356,313.33	81,314	49.24	14.51	61.80
0.1	MySQL CE v8.0.38	AES-256	34,870.00	2,211	32.77	41.80	7.61
0.1	MySQL CE v8.0.38	no-encryption	72,450.00	22,204	33.40	33.84	47.92
1	MySQL CE v8.0.38	AES-256	2,032,900.00	88,874	35.30	21.64	56.49

SF	Database	Encryption algorithm	Execution Average (ms)	StdDev Average (ms)	RAM (%)	CPU (%)	Disk Time (%)
1	MySQL CE v8.0.38	no-encryption	1,950,586.67	41,949	33.21	19.94	59.56
0.1	MySQL EE v9.2	AES-256	10,823	1,704	32.26	21.87	71.49
0.1	MySQL EE v9.2	no-encryption	9,913	386	44.17	49.82	24.60
1	MySQL EE v9.2	AES-256	1,094,980	7,864	39.68	17.02	71.37
1	MySQL EE v9.2	no-encryption	901,453	167,945	36.69	16.79	63.90
0.1	SQL Server	AES-128	7,229.67	247	55.03	41.22	0.58
0.1	SQL Server	AES-192	5,408.67	1,360	37.80	57.17	0.94
0.1	SQL Server	AES-256	6,419.00	887	37.06	46.17	1.40
0.1	SQL Server	no-encryption	7,764.33	424	55.14	30.09	0.50
1	SQL Server	AES-128	18,151.33	2,593	54.98	54.81	0.79
1	SQL Server	AES-192	19,293.00	1,493	54.73	53.81	0.64
1	SQL Server	AES-256	20,392.67	3,389	42.65	38.24	3.11
1	SQL Server	no-encryption	19,034.67	765	50.48	39.05	0.32
10	SQL Server	AES-128	177,422.00	8,637	95.08	88.42	79.70
10	SQL Server	AES-192	175,735.67	4,118	95.72	93.45	107.49
10	SQL Server	AES-256	202,293.00	21,658	93.27	74.64	49.70
10	SQL Server	no-encryption	159,360.33	6,433	82.60	94.14	19.58

SQL Server for Scale Factor 0.1 shows that AES-128, AES-192, and AES-256 perform better than the no-encryption option. This may suggest inefficient memory management for the no-encryption setup or reflect the high variability often observed at small factors like 0.1. Table 15 shows that non-encryption requires more memory, with an overhead of 25.4% compared to encryption algorithms. Future studies could investigate Disk, CPU and RAM usage on a per-query basis to identify which queries could have caused the non-encryption setup to underperform. This can give another perspective on how the system deals with multiple queries and which ones can impact with the execution time performance.

At Scale Factor 1, there's an increase of 145.1% compared to scale factor 0.1 (non-encryption values), along with a decrease of 8.45% RAM usage. The disk usage decreased by 36% compared to scale factor 0.1. However, disk usage spiked at 8.34% in one AES-256 test, and AES-256 showed the slowest execution time among encryption algorithms at this scale. Additionally, AES-256 had the highest standard deviation (3,389 ms), reflecting higher execution variability compared to AES-128 (2,593 ms) and AES-192 (1,493 ms), which may indicate performance instability at this scale. Calculating the execution query time overhead when moving from Scale Factor 0.1 to Scale Factor 1 gives a 2.45-fold increase difference in the non-encrypted database. The expected result was a 10-fold increase, because the size from scale factor 0.1 to 1 is 10 times more, indicating that the performance remains relatively efficient and does not degrade when scaling.

Scale Factor 10 in terms of total execution times, the Encryption algorithms underperformed against no-encryption as expected, but AES-192 performed 0.95% faster than AES-128, which was unexpected. However, AES-128 showed greater variability with a standard deviation of 8,637 ms compared to just 4,118 ms for AES-192, reinforcing the stability of AES-192 at this larger scale. AES-256 was both slower and more inconsistent, with the highest standard deviation of 21,658 ms. This likely reflects a combination of hardware-level optimizations and resource management differences at higher loads. Calculating the execution times overhead when moving from 19,034.67 ms at Scale Factor 1 (no-encryption) to 159,360.33 ms at Scale Factor 10, gives an 8.37x increase, aligning with expectations. This increase suggests that, although performance scales with data size, it begins to stabilize as the scale grows with potential for underperformance, taking much more time than expected to query from scale factor 100 to scale factor 10, for example. Future studies with a higher scale factor, such as scale factor 100, could confirm this pattern and explore its impact on system resource consumption.

To finalize the analysis of SQL Server results is possible to say that the overhead between scale factors was expected when scaling, having a better performance when comparing scale factor to scale factor 0.1. It is needed to scale much more to see proper results because it is possible to see unexpected results when comparing encryption algorithms against a no-encryption baseline, which can suggest a lower scale

factor to be able to compare or poor memory management and CPU and RAM usage as seen in scale factor where no-encryption used 2.92% more RAM than the average of encrypted setups. When assessing performance among scale factors, AES-192 consistently emerges as the most efficient encryption algorithm, outperforming others at Scale Factor 0.1 and Scale Factor 10, only underperforming in Scale Factor 1, where AES-128 outperforms AES-192 slightly. While, AES-192 performs best at scale factors 0.1 and 10, AES-128 shows better performance at scale factor 1. Overall, AES-192 is more consistent across different scale factors.

For MySQL at Scale Factor 0.1, AES-256 is 2.08 faster than the no-encryption setup, meaning that AES-256 executes more than twice as fast as the no-encryption test runs. This can be attributed to high disk usage in the no-encryption setup; as disk usage increases, having an increase of 529.70% difference in disk usage between no-encryption and AES-256, the system spends more time on disk reads, creating I/O bottlenecks that likely slowdown query performance.

At Scale Factor 1, AES-256 takes longer to execute than the no-encryption setup. This is reflected in the higher standard deviation of AES-256 (88,874 ms) compared to 41,949 ms for no-encryption, suggesting more execution time fluctuation for encrypted queries. With an overhead of 1.04, AES-256 is slightly slower than no-encryption. However, the small gap suggests performance is stabilizing when scaling. Testing at even higher Scale Factors could reveal whether encryption performance improves compared to no-encryption.

Overall, SQL Server seems better equipped to handle complex queries and large datasets across different Scale Factors. At the same time, MySQL, using TDE AES-256, tends to underperform at higher scale factors compared to the no-encryption baseline, although it might outperform no-encryption in smaller scale factors. Based on the results, TDE AES-256 in MySQL may not be recommended for optimal performance.

For Oracle at scale factor 0.1, no-encryption outperforms AES-192 and AES-256. Disk usage for AES-192 is 4260.95% higher than no-encryption and Disk usage for AES-256 is 212.5% higher than no-encryption, which shows that no-encryption has lower computational overhead but exhibits higher RAM usage, suggesting inefficiencies in resource handling. AES-192, while secure, suffers from higher disk dependency, which impacts performance. No-encryption, though computationally lighter, exhibits inefficiencies in memory and disk management, resulting in less predictable performance.

For Oracle at scale factor 1, AES-256 outperformed all other AES key sizes, showing less than 24.9% to query compared to AES-192 and a 31.3% less time to query compared to AES-128. Even though no-encryption had the best performance overall compared to the encrypted test cases, it showed an higher disk usage with an average of 73.19% compared to only 31.02% with AES-256. CPU utilization had its major use under AES-128 with 32.21%, close to no-encryption scenario with 31.38% where AES-192 required less CPU with 19.06%. AES-256 was the best encryption

algorithm key size, balancing the resource usage, offering strong encrypted data and a stable performance under scale factor 1.

For Oracle at scale factor 10, no-encryption outperformed AES-128, AES-192, and AES-256, as expected, showcasing that when scaling the database size, it's possible to see more stable results, when it's expected to see the total execution time by the same order of key size of the encryption algorithms used. AES-128 outperformed AES-192, and AES-256. AES-192 took 21.83% longer to execute than AES-128, and AES-256 compared to AES-128, took 427.2% longer to query. In terms of system resources, AES-192 used 8.83% less RAM than AES-128 but relies on disk 4.42% more than AES-128. AES-256, while the most secure, suffers from significant computational overhead, uses 21.91% more RAM than no-encryption but decreased disk usage by 31.44% compared to no-encryption. AES-128 shows as a solution for larger databases, where outperformed the other encryption algorithms by either operative system performance or execution time performance being an optimal choice for scale factor 10.

In MySQL Enterprise v9.2, significant performance differences were observed between Scale Factors 0.1 and 1. At scale factor 0.1, is shown different results when comparing AES-256 to no-encryption. In some queries such as Q3, Q7, Q9 and Q13 it takes 1.2 to 1.4 times longer than no-encryption to execute the query. However, there is some execution times where AES-256 outperforms the no-encryption execution times, likely due to caching effects or execution plans under light loads. At scale factor 1, in contrast, the performance impact of AES-256 is more noticeable, while the powertest shows a slight increase in the total execution, the throughput shows a significant overhead, ranging from 30% to 40% longer execution times when comparing AES-256 to no-encryption, this overhead is given to a larger database, additional CPU cycles and a higher waiting time in disk for I/O demands. For AES-256 compared to no-encryption, there was an increase from 16% to 25% in CPU values, while in some other tests CPU usage dropped to 12.92% approximately. Disk Time goes from 54.9% to 71.97% comparing no-encryption to AES-256 respectively, reflecting the additional I/O demands imposed by encryption. In terms of RAM, is variable in AES-256, showing that it can go from 31.76% to 43.64% in an encrypted state, showing that memory demands can fluctuate depending on the workload and caching behaviour of the execution plans in the throughput when comparing powertest to throughput, when executing a throughput we have multiple executions of queries and each throughput test includes 22 queries, executed in different orders, affecting caching and performance. This analysis of RAM, CPU and Disk time indicates that as data volume increases, additional CPU cycles for encryption and higher disk I/O demands have impact on the performance.

7 CONCLUSIONS

In this thesis, we explored the impact of various encryption algorithms on performance metrics across different database platforms (Oracle, MySQL, and SQL Server) and under different load factors. The results demonstrate that the choice of encryption algorithm significantly impacts execution time, CPU usage, RAM consumption, and disk time.

The experimental evaluation explored the impact of the TDE algorithms on database platforms Oracle, MySQL, and SQL Server. At the same time, the effect on the hardware was analyzed using performance metrics such as CPU usage, RAM consumption, and disk time.

The Oracle system showed AES-192 as its most balanced encryption algorithm when evaluating different scale factors. AES-192 delivered a better performance than AES-128 at both 0.1 and 1 scale factors, while in scale factor 1 AES-256 was the best encryption algorithm. AES-128 delivered a better performance than AES-192 and AES-256 when the workload reached scale factor 10. It also showed the lowest standard deviation (12,470 ms), while AES-192 and AES-256 had much higher variability with 176,137 ms and 2,990,971 ms respectively. The no-encryption configuration in Oracle outperformed the encrypted setups during scale factor 0.1, while in scale factor 1, the no-encryption as expected outperformed the encryption algorithm test cases. From scale factors 1 to 10, AES-128, AES-192, and AES-256 took 1.55x, 1.99x, 10.76x more in scale factor 10, showing a modest stabilization. The execution time stabilized when the workload reached scale factor 10, indicating that large datasets produce consistent performance results in Oracle regardless of encryption levels.

For SQL Server, AES-192 was the most balanced TDE algorithm in almost every scale factor, except for the scale factor 1, where it is possible to see for AES-128 performing better with an average of 18,151 ms compared to AES-192 average of 19,293 ms and AES-256 average of 20,393 ms. Standard deviation further supports this, where AES-128 had 2,593 ms, compared to AES-192 (1,493 ms) and AES-256 (3,389 ms). AES-256, while being the strongest in terms of key size, as expected, had higher execution times and increased disk time, making it less suitable for performance-sensitive workloads. At Scale Factor 1, CPU usage for AES-256 slightly decreased 38.24% compared to Scale Factor 0.1, while RAM usage increased by 15%. Disk time remains steady across factors, though AES-256 experiences a spike, making it the slowest for total execution time. The execution time overhead from Scale Factor 0.1 to 1 is 2.45-fold, in the non-encrypted database, which is favorable compared to the expected 10-fold increase, indicating stable performance with scaling. At Scale Factor 10, encryption algorithms performed slower than no-encryption as expected; however, AES-192 was slightly faster than AES-128 by 0.95% lower average execution time, likely due to hardware optimizations and resource management at high loads. It also had the lowest standard deviation of all encrypted key sizes at

this scale with 4,118 ms compared to 8,637 ms for AES-128 and 21,658 ms for AES-256, showing more consistent execution. The execution time overhead moving from Scale Factor 1 to 10 is 8.37-fold, in the non-encrypted database, indicating performance stabilizes as data size grows, with potential underperformance as scale increases.

Although AES supports multiple key sizes, MySQL's tablespace encryption and TDE configuration only expose AES-256, with no available or documented support for AES-128 and AES-192 in the tested versions.

At Scale Factor 0.1, AES-256 outperformed the no-encryption setup, which took twice as long due to high disk usage, with a 108% overhead. This was reflected in the standard deviation as well: 2,211 ms for AES-256 versus 22,204 ms for no-encryption, indicating far less variability in encrypted runs. However, at Scale Factor 1, the performance reversed, with AES-256 taking slightly longer than the no-encryption setup, showing an overhead score of 1.04. This suggests that as the data size increases, MySQL's performance with AES-256 remains relatively stable, though slightly slower.

In the MySQL experiments, MySQL Enterprise v9.2 can manage encryption overhead more stably in both scale factors compared to MySQL version 8.0.38, which shows significant variability and occasional spikes in overhead.

SQL Server consistently shows a strong capacity to handle complex queries and larger datasets across different Scale Factors. While MySQL AES-256 tends to have a better performance compared to no-encryption at smaller scales, when scaling, it tends to stabilize or underperform. MySQL AES-256 may be better prepared to execute basic queries on smaller datasets. SQL Server is more capable of handling larger performance data workloads. At scale factor 10, a more stable factor with more realistic workload, SQL Server outperformed Oracle, in the three different key sizes of AES, where AES-128 and AES-192, Oracle's total execution time was nearly 9 to 10 times slower than SQL Server and Oracle's AES-256 was nearly forty times higher than SQL Server, based on average total execution time. No-encryption performance for Oracle was still higher than SQL Server's no-encryption. SQL Server showed consistent performance, regardless of the encryption level, where Oracle's performance was negatively impacted across all configurations, showing that SQL Server performance is much more efficient than Oracle performance, in terms of scalability. Saying this, it is possible to claim that SQL Server is the best database to be used, and Oracle follows next, while MySQL had higher variations in terms of results and it had a poor performance compared to Oracle and SQL Server. The author in [46], concludes saying Oracle and SQL Server are much faster and a safer option to be used when compared to MySQL. This is consistent with the findings in this thesis, where Oracle and SQL Server outperformed and showed more consistency than MySQL v8.0.38. However, MySQL v9.2 got more consistent results than its predecessor, but still underperformed when compared to SQL Server and Oracle execution test runs.

Across all execution tests with different Scale Factors, different key sizes for AES, and different databases, Oracle under heavier Scale Factors showed higher execution time overheads, suggesting the impact of encryption can vary between database systems and Scale Factors. However, the overhead remained modest for SQL Server and for MySQL at smaller scales, which is also reflected in [11] where the encryption method showed minor overhead, preserving its transparent nature.

Future work should include increasing the scale factors to analyze how encryption affects Oracle, MySQL and SQL Server performance through system resource monitoring and execution time measurement. The performance stabilized at Scale Factor 1 for MySQL and Scale Factor 10 for both SQL Server and Oracle, which indicates that higher scale factors would produce more consistent results with fewer variations and longer query times for encrypted databases. A dedicated machine combined with background process isolation would produce more precise and consistent results because background processes create variations in the results. The observed high standard deviations, particularly at lower scale factors and under AES-256 encryption, highlight the need for background process isolation and dedicated test environments to minimize measurement noise. The analysis would provide a better understanding of the encryption strength on system performance and system efficiency. The research should also analyze different encryption algorithms to evaluate their individual performance characteristics and compare their results to other encryption algorithms. As well as perform real-world scenarios in cloud environments, to have on-premises and cloud environment performance testing to evaluate modern tools and their behaviour with modern database tools.

REFERENCES

[1] Ö. Aslan, S. S. Aktuğ, M. Ozkan-Okay, A. A. Yilmaz, and E. Akin, "A comprehensive review of cyber security vulnerabilities, threats, attacks, and solutions," *Electronics (Basel)*, vol. 12, no. 6, p. 1333, 2023.

[2] R. Santos, "Enhancing Data Security in Data Warehousing," PhD Thesis, Coimbra, 2014. Accessed: Apr. 24, 2025. [Online]. Available: http://hdl.handle.net/10316/25230

[3] E. D. Madyatmadja, A. N. Hakim, and D. J. M. Sembiring, "Performance testing on Transparent Data Encryption for SQL Server's reliability and efficiency," *J. Big Data*, vol. 8, no. 1, p. 134, 2021, doi: 10.1186/S40537-021-00520-Z.

[4] R. J. Santos, J. Bernardino, and M. Vieira, "Evaluating the feasibility issues of data confidentiality solutions from a data warehousing perspective," in *Data Warehousing and Knowledge Discovery: 14th International Conference, DaWaK 2012, Vienna, Austria, September 3-6, 2012. Proceedings 14*, A. and D. U. Cuzzocrea, Ed., Vienna, Austria: Springer Berlin Heidelberg, 2012, pp. 404–416. doi: 10.1007/978-3-642-32584-7_33.

[5] A. Abdullah, "Advanced Encryption Standard (AES) Algorithm to Encrypt and Decrypt Data," Aug. 2017.

[6] O. Corporation, "InnoDB Data-at-Rest Encryption." Accessed: Oct. 28, 2024. [Online]. Available: https://dev.mysql.com/doc/refman/8.4/en/faqs-tablespace-encryption.html#faq-tablespace-encryption-access

[7] "DB-Engines Ranking - popularity ranking of database management systems." [Online]. Available: https://db-engines.com/en/ranking

[8] J. Monteiro, F. Sá, and J. Bernardino, "Experimental Evaluation of Graph Databases: JanusGraph, Nebula Graph, Neo4j, and TigerGraph," *Applied Sciences 2023, Vol. 13, Page 5770*, vol. 13, no. 9, p. 5770, May 2023, doi: 10.3390/APP13095770.

[9] E. Pina, F. Sá, and J. Bernardino, "NewSQL Databases Assessment: CockroachDB, MariaDB Xpand, and VoltDB," *Future Internet*, vol. 15, no. 1, p. 10, Dec. 2022, doi: 10.3390/FI15010010.

[10] S. Istifan and M. Makovac, "Performance benchmarking of data-at-rest encryption in relational databases," Bachelor Thesis, 2022.

[11] F. Moulianitakis and K. Asimakopoulos, "Benchmarking Framework for Transparent Data Encryption Systems," Luleå University of Technology, Department of Computer Science, Electrical and Space Engineering, 2019.

[12] G. Wessler Boneti, "Identify threat model and protection offered by different encryption possibilities in MySQL, PostgreSQL and Oracle," 2023.

[13] O. Kara, "Lower data attacks on Advanced Encryption Standard," *Turkish J. Electr. Eng. Comput. Sci.*, vol. 32, no. 2, pp. 338–357, 2024, doi: 10.55730/1300-0632.4072.

[14] B. Li, M. Xu, Y. Zhou, H. Liu, and R. Zhang, "Optimization of Security Identification in Power Grid Data through Advanced Encryption Standard Algorithm," *Journal of Cyber Security and Mobility*, pp. 239–264, 2024.

[15] K. Assa-Agyei, K. Owa, F. Olajide, and T. Al Hadhrami, "A Multi-Chaotic Key Expansion for Advanced Encryption Standard (AES) Algorithm," in *International Conference on Computing, Networking and Communications, ICNC 2024, Big Island, HI, USA, February 19-22, 2024*, IEEE, 2024, pp. 1–7. doi: 10.1109/ICNC59896.2024.10556263.

[16] M. F. Mushtaq, S. Jamel, A. H. Disina, Z. A. Pindar, N. S. A. Shakir, and M. M. Deris, "A survey on the cryptographic encryption algorithms," *International Journal of Advanced Computer Science and Applications*, vol. 8, no. 11, 2017.

[17] A. Hamza and B. Kumar, "A Review Paper on DES, AES, RSA Encryption Standards," in *2020 9th International Conference System Modeling and Advancement in Research Trends (SMART)*, Dec. 2020, pp. 333–338. doi: 10.1109/SMART50582.2020.9336800.

[18] A. Biryukov and C. De Cannière, "Data encryption standard (DES)," *Encyclopedia of Cryptography and Security*, pp. 295–301, 2011.

[19] S. Srilaya and S. Velampalli, "Performance Evaluation for DES and AES Algorithms- An Comprehensive Overview," in *2018 3rd IEEE International Conference on Recent Trends in Electronics, Information & Communication Technology (RTEICT)*, May 2018, pp. 1264–1270. doi: 10.1109/RTEICT42901.2018.9012536.

[20] S. Stachowiak, M. Kurkowski, and A. Soboń, "SAT vs. Substitution Boxes of DES like Ciphers," in *2021 IEEE 30th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, Oct. 2021, pp. 113–118. doi: 10.1109/WETICE53228.2021.00032.

[21] H. O. Alanazi, B. B. Zaidan, A. A. Zaidan, H. A. Jalab, M. Shabbir, and Y. Al-Nabhani, "New Comparative Study Between DES, 3DES and AES within Nine Factors," vol. 2, 2010.

[22] P. Patil, P. Narayankar, D. G. Narayan, and S. M. Meena, "A Comprehensive Evaluation of Cryptographic Algorithms: DES, 3DES, AES, RSA and Blowfish," *Procedia Comput Sci*, vol. 78, pp. 617–624, Jan. 2016, doi: 10.1016/J.PROCS.2016.02.108.

[23] Z. Hercigonja, "Comparative Analysis of Cryptographic Algorithms," *International Journal of DIGITAL TECHNOLOGY & ECONOMY*, vol. 1, no. 2, 2016.

[24] O. Srinivasa Rao, "Performance Analysis of DES and Triple DES," *Int J Comput Appl*, vol. 130, no. 14, pp. 975–8887, 2015.

[25] R. Ramya Devi and V. Vijaya Chamundeeswari, "Triple DES: privacy preserving in big data healthcare," *Int J Parallel Program*, vol. 48, no. 3, pp. 515–533, 2020.

[26] D. Buell, "Modern symmetric ciphers—Des and Aes," in *Fundamentals of Cryptography: Introducing Mathematical and Algorithmic Foundations*, Springer, 2021, pp. 123–147.

[27] C. Boura, P. Derbez, and M. Funk, "Related-Key Differential Analysis of the AES," *IACR Transactions on Symmetric Cryptology*, vol. 2023, no. 4, pp. 215–243, 2023.

[28] "Federal Information Processing Standards Publication 197 Announcing the ADVANCED ENCRYPTION STANDARD (AES)," 2001, [Online]. Available: http://csrc.nist.gov/csor/

[29] J. Daemen and V. Rijmen, "AES Proposal: Rijndael," *Mathematics, Computer Science*, 1998.

[30] Oracle White Paper, "Oracle Advanced Security Transparent Data Encryption Best Practices," 2012.

[31] Microsoft, "Transparent Data Encryption (TDE)," 2024. [Online]. Available: https://learn.microsoft.com/en-us/sql/relational-databases/security/encryption/transparent-data-encryption?view=sql-server-ver16

[32] R. Coles Michael and Landrum, "Transparent Data Encryption," in *Expert SQL Server 2008 Encryption*, Berkeley, CA: Apress, 2009, pp. 127–150. doi: 10.1007/978-1-4302-3365-7_6.

[33] "MySQL :: Announcing: MySQL Enterprise Transparent Data Encryption (TDE)." [Online]. Available: https://forums.mysql.com/read.php?3,644914

[34] MySQL, "Transparent Data Encryption (TDE) in MySQL Enterprise Edition." [Online]. Available: https://www.mysql.com/products/enterprise/tde.html

[35] "Introduction to Transparent Data Encryption." Accessed: Apr. 29, 2025. [Online]. Available: https://docs.oracle.com/en/database/oracle/oracle-database/21/asoag/introduction-to-transparent-data-encryption.html#GUID-688B2CA5-EB00-4BEE-9486-9046670CCA70

[36] "MySQL :: MySQL : InnoDB Transparent Tablespace Encryption." Accessed: Apr. 29, 2025. [Online]. Available: https://dev.mysql.com/blog-archive/mysql-innodb-transparent-tablespace-encryption/

[37] O. Corporation, "InnoDB Data-at-Rest Encryption." [Online]. Available: https://dev.mysql.com/doc/refman/8.0/en/innodb-data-encryption.html

[38] PostgreSQL, "Encryption Options in PostgreSQL," 2024. [Online]. Available: https://www.postgresql.org/docs/current/encryption-options.html

[39] PostgreSQL, "pgcrypto Module Documentation." [Online]. Available: https://www.postgresql.org/docs/current/pgcrypto.html

[40] Microsoft, "Always Encrypted," 2024. [Online]. Available: https://learn.microsoft.com/en-us/sql/relational-databases/security/encryption/always-encrypted-database-engine?view=sql-server-ver16

[41] Microsoft, "Encrypt a Column of Data," 2024. [Online]. Available: https://learn.microsoft.com/en-us/sql/relational-databases/security/encryption/encrypt-a-column-of-data?view=sql-server-ver16

[42] O. Corporation, "Encrypting Tablespaces: A Transparent Data Encryption Guide," 2024. [Online]. Available: https://blogs.oracle.com/connect/post/encrypting-Tablespaces

[43] M. Corporation, "Encryption, hashing, and compression functions," 2024. [Online]. Available: https://mariadb.com/kb/en/encryption-hashing-and-compression-functions/

[44] R. J. Santos, D. Rasteiro, J. Bernardino, and M. Vieira, "A specific encryption solution for data warehouses," in *Database Systems for Advanced Applications: 18th International Conference, DASFAA 2013, Wuhan, China, April 22-25, 2013. Proceedings, Part II 18*, 2013, pp. 84–98.

[45] V. Lukaj, A. Catalfamo, F. Martella, M. Fazio, M. Villari, and A. Celesti, "A NoSQL DBMS Transparent Data Encryption Approach for Cloud/Edge Continuum," in *2023 IEEE Symposium on Computers and Communications (ISCC)*, 2023, pp. 430–435. doi: 10.1109/ISCC58397.2023.10217933.

[46] D. L. Hussein, "Evaluating Aggregate Functions and Machine Learning Integration: A Comparative Analysis of Performance, Security, and NoSQL Connectivity in Oracle, SQL Server, and MySQL," *UHD Journal of Science and Technology*, vol. 8, no. 2, pp. 7–23, 2024.

[47] E. G. AbdAllah, Y. R. Kuang, and C. Huang, "Advanced encryption standard new instructions (aes-ni) analysis: Security, performance, and power consumption," in *Proceedings of the 2020 12th International Conference on Computer and Automation Engineering*, 2020, pp. 167–172.

[48] R. Dzer and R. Dima, "ENHANCING QUERY PERFORMANCE AND PRIVACY ON RELATIONAL CLOUD DATABASE," 2015.

[49] L. S. Gubala Hari Babu and S. N. S. Dodla, "Comparative Analysis of Oracle and MySQL Databases : A Study on Query Execution and Scalability," Bachelor Thesis, Blekinge Institute of Technology, Department of Computer Science, 2024.

[50] A. T. Katsadouris, "Advanced Cryptographic Techniques For Database Security," MASTER THESIS, NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS, Athens, 2021.

[51] S. Ansmink, "Encrypted Query Processing in DuckDB," Master Thesis, Vrije Universiteit Amsterdam, 2021.

[52] V. Sidorov and W. K. Ng, "Transparent Data Encryption for Data-in-Use and Data-at-Rest in a Cloud-Based Database-as-a-Service Solution," in *2015 IEEE World Congress on Services*, 2015, pp. 221–228. doi: 10.1109/SERVICES.2015.40.

[53] K. Natarajan and V. Shaik, "Transparent Data Encryption: Comparative Analysis and Performance Evaluation of Oracle Databases," in *2020 Fifth International Conference on Research in Computational Intelligence and Communication Networks (ICRCICN)*, 2020, pp. 137–142. doi: 10.1109/ICRCICN50933.2020.9296168.

[54] J. Hu and A. Klein, "A Benchmark of Transparent Data Encryption for Migration of Web Applications in the Cloud," in *2009 Eighth IEEE International Conference on*

Dependable, Autonomic and Secure Computing, 2009, pp. 735–740. doi: 10.1109/DASC.2009.85.

[55] U. T. Mattsson, "Database Encryption - How to Balance Security with Performance," *SSRN Electronic Journal*, Feb. 2005, doi: 10.2139/SSRN.670561.

[56] E. Shmueli, R. Vaisenberg, E. Gudes, and Y. Elovici, "Implementing a Database Encryption Solution, Design and Implementation Issues," *Comput Secur*, 2014, doi: https://doi.org/10.1016/j.cose.2014.03.011.

[57] P. Colbert and B. Juliano, "A Performance Comparison of Microsoft SQL Server 2008 Transparent Data Encryption and NetLib Encryptionizer.," in *Conference: Proceedings of the 2009 International Conference on Security & Managemen*, Apr. 2009, pp. 93–98.

[58] D. Bărbulescu, A.-C. Enache-Ducoffe, and M. Togan, "A new comparative study of database security," *Romanian Journal of Information Technology and Automatic Control*, 2023, doi: 10.33436/v33i3y202302.

[59] Transaction Processing Performance Council, "TPC BENCHMARKTM H (Decision Support) Standard Specification Revision 3.0.1." [Online]. Available: https://www.tpc.org/tpc_documents_current_versions/current_specifications5.asp

[60] R. Moussa, "TPC-H Benchmark Analytics Scenarios and Performances on Hadoop Data Clouds," in *Communications in Computer and Information Science*, Oct. 2012, pp. 220–234. doi: 10.1007/978-3-642-30507-8_20.

[61] A. Thanopoulou, P. Carreira, and H. Galhardas, "Benchmarking with TPC-H on Off-the-Shelf Hardware: An Experiments Report," in *Proc. of the 14th Int'l Conference on Enterprise Information Systems (ICEIS'2012)*, Jan. 2012.

[62] A. Akhtar, "Popularity ranking of database management systems," *arXiv preprint arXiv:2301.00847*, 2023, doi: https://doi.org/10.48550/arXiv.2301.00847.

[63] L. Pasarin, P. Advisor, E. M. Sarroca, and P. C. Arias, "Analysis of alternatives to store genealogical trees using Graph Databases," Master Thesis, Universitat Politècnica de Catalunya, 2013.

[64] "Oracle Database Editions." [Online]. Available: https://docs.oracle.com/cd/E11882_01/license.112/e47877/editions.htm#DBLIC134

[65] X. Yang, "Analysis of DBMS: MySQL Vs PostgreSQL," Bachelor Thesis, Kemi-Tornio University of Applied Sciences, 2011.

[66] J. I. Janjua, T. A. Khan, S. Zulfiqar, and M. Q. Usman, "An Architecture of MySQL Storage Engines to Increase the Resource Utilization," in *2022 International Balkan Conference on Communications and Networking (BalkanCom)*, 2022, pp. 68–72.

[67] M. Ilić, L. Kopanja, D. Zlatković, M. Trajković, and D. Ćurguz, "Microsoft sql server and oracle: Comparative performance analysis," in *The 7th International conference Knowledge management and informatics*, 2021, pp. 33–40.

[68] M. C. Richards, "AES: The making of a new encryption standard," *SANS Institute InfoSec*, 2001.

[69] Oracle, "MySQL 8.4 Reference Manual - FAQs: Tablespace Encryption," 2024. [Online]. Available: https://dev.mysql.com/doc/refman/8.4/en/faqs-tablespace-encryption.html#faq-tablespace-encryption-access

[70] S. Duraiswamy *et al.*, "General Considerations of Using Transparent Data Encryption".

APPENDIX A - PYTHON SCRIPT MULTI-QUERY EXECUTION CREATION

```
# Join all queries in one file - Throughput Test
def ReturnMySQLThroughputTest():
    j = 0
    # with open(files.destiny+'ThroughputMySQLTest1-40.sql', 'a') as fileToWrite:
        # fileToWrite.write(files.timeToShow)
        # fileToWrite.write(files.setStatisticsOn)
    for x in files.ThroughputTest:
        if(j==files.sizeOfThroughput):
            break;
        j = j+1
        with open(files.destiny+'ThroughputMySQLTest1-40.sql', 'a') as fileToWrite:
            fileToWrite.write(
                "-- ThroughputTest-"+str(j)+"-begin\n\n")
        querynumber = 0
        for y in x:
            with open(files.path+str(y)+'.sql', 'r') as file:
                # read content from first file
                with open(files.destiny+'ThroughputMySQLTest1-40.sql', 'a') as
fileToWrite:
                    if (j == 1 and y == 0):
                        for line in file:
                            # append content to second file
                            fileToWrite.write(line)
                    elif (y != 0):
                        if y in files.to_include:
                            fileToWrite.write(
                                "\n\n-- Query "+str(y)+"-Begin\n\n")
                            for line in file:
```

```
                    # append content to second file
                    fileToWrite.write(line)
                fileToWrite.write(
                    "\n\n-- Query "+str(y)+"-End\n")
            elif not files.to_include:
                fileToWrite.write(
                    "\n\n-- Query "+str(y)+"-Begin\n\n")
                for line in file:
                    # append content to second file
                    fileToWrite.write(line)
                fileToWrite.write(
                    "\n\n-- Query "+str(y)+"-End\n")
        else:
            continue
    with open(files.destiny+'ThroughputMySQLTest1-40.sql', 'a') as fileToWrite:
        fileToWrite.write("\n\n\n-- ThroughputTest-"+str(j)+" -End\n\n\n")
with open(files.destiny+'ThroughputMySQLTest1-40.sql', 'a') as fileToWrite:
    fileToWrite.write("\n\nDROP VIEW REVENUE0;\n")
    # fileToWrite.write(files.timeToShow)


# Power Test MySql
def ReturnPowerTestMySql():
    j = 0

    with open(files.destiny+'PowerTestMySql.sql', 'a') as fileToWrite:
        # fileToWrite.write(files.timeToShow)
        # fileToWrite.write(files.setStatisticsOn)
        fileToWrite.write(
            "-- Start of the power test-begin\n\n")
    for x in files.powertest:
        j = j+1
```

```
with open(files.destiny+'PowerTestMySql.sql', 'a') as fileToWrite:
    fileToWrite.write("-- PowerTest-" +str(j)+"-begin\n\n")
querynumber = 0
for y in x:
    if (y == 0 and j == 1):
        querynumber = 0
    else:
        querynumber = querynumber+1

    with open(files.path+str(y)+'.sql', 'r') as file:
        # read content from first file
        with open(files.destiny+'PowerTestMySql.sql', 'a') as fileToWrite:
            if (y == 0 and j == 1):
                for line in file:
                    # append content to second file
                    fileToWrite.write(line)
            elif (y != 0):
                if y in files.to_include:
                    fileToWrite.write(
                        "\n\n-- Query "+str(y)+"-Begin)\n\n")
                    for line in file:
                        # append content to second file
                        fileToWrite.write(line)
                    fileToWrite.write(
                        "\n\n-- Query "+str(y)+"-End\n")
                elif not files.to_include:
                    fileToWrite.write(
                        "\n\n-- Query "+str(y)+"-Begin\n\n")
                    for line in file:
                        # append content to second file
                        fileToWrite.write(line)
                    fileToWrite.write(
```

```
"\n\n-- Query "+str(y)+"-End\n")

    with open(files.destiny+'PowerTestMySql.sql', 'a') as fileToWrite:
        fileToWrite.write(
            "-- PowerTest-End"+'\n\n\n')
        # fileToWrite.write(files.setStatisticsOff)
        # fileToWrite.write(files.timeToShow)


# Power Test Sql Server
def ReturnPowerTest():
    j = 0

    with open(files.destiny+"powertest", 'a') as fileToWrite:
        fileToWrite.write(files.timeToShow)
        fileToWrite.write(files.setStatisticsOn)
        fileToWrite.write(
            "print('-----------Start of the power test-begin-----------')\n\n")
    for x in files.powertest:
        j = j+1
        with open(files.destiny+"powertest", 'a') as fileToWrite:
            fileToWrite.write("print('-----------PowerTest-"       +str(j)+"-begin-----------
')\n\n")
        querynumber = 0
        for y in x:
            if (y == 0 and j == 1):
                querynumber = 0
            else:
                querynumber = querynumber+1

            with open(files.path+str(y)+'.sql', 'r') as file:
                # read content from first file
                with open(files.destiny+"powertest", 'a') as fileToWrite:
```

```
                if (y == 0 and j == 1):
                    for line in file:
                        # append content to second file
                        fileToWrite.write(line)
                elif (y != 0):
                    if y in files.to_include:
                        fileToWrite.write(
                            "\n\nprint('-----------Query "+str(y)+"-Begin-----------')\n\n")
                        for line in file:
                            # append content to second file
                            fileToWrite.write(line)
                        fileToWrite.write(
                            "\n\nprint('-----------Query "+str(y)+"-End-----------')\n")
                    elif not files.to_include:
                        fileToWrite.write(
                            "\n\nprint('-----------Query "+str(y)+"-Begin-----------')\n\n")
                        for line in file:
                            # append content to second file
                            fileToWrite.write(line)
                        fileToWrite.write(
                            "\n\nprint('-----------Query "+str(y)+"-End-----------')\n")

    with open(files.destiny+"powertest", 'a') as fileToWrite:
        fileToWrite.write(
            "print('-----------PowerTest-End-----------')"+'\n\n\n')
        fileToWrite.write(files.setStatisticsOff)
        fileToWrite.write(files.timeToShow)

# Join all queries in one file - Throughput Test
def ReturnThroughputTest():
    j = 0
    with open(files.destiny+"powertest", 'a') as fileToWrite:
```

```
    fileToWrite.write(files.timeToShow)
    fileToWrite.write(files.setStatisticsOn)
for x in files.ThroughputTest:
    if(j==files.sizeOfThroughput):
        break;
    j = j+1
    with open(files.destiny+"powertest", 'a') as fileToWrite:
        fileToWrite.write(
            "print('-----------ThroughputTest-"+str(j)+"-begin-----------')\n\n")
    querynumber = 0
    for y in x:
        with open(files.path+str(y)+'.sql', 'r') as file:
            # read content from first file
            with open(files.destiny+"powertest", 'a') as fileToWrite:
                if (j == 1 and y == 0):
                    for line in file:
                        # append content to second file
                        fileToWrite.write(line)
                elif (y != 0):
                    if y in files.to_include:
                        fileToWrite.write(
                            "\n\nprint('-----------Query "+str(y)+"-Begin-----------')\n\n")
                        for line in file:
                            # append content to second file
                            fileToWrite.write(line)
                        fileToWrite.write(
                            "\n\nprint('-----------Query "+str(y)+"-End-----------')\n")
                    elif not files.to_include:
                        fileToWrite.write(
                            "\n\nprint('-----------Query "+str(y)+"-Begin-----------')\n\n")
                        for line in file:
                            # append content to second file
```

```
                    fileToWrite.write(line)
                fileToWrite.write(
                    "\n\nprint('-----------Query "+str(y)+"-End-----------')\n")
            else:
                continue
    with open(files.destiny+"powertest", 'a') as fileToWrite:
        fileToWrite.write("\n\n\nprint('-----------ThroughputTest-"+str(j)+"        -
End-----------')\n\n\n")
    with open(files.destiny+"powertest", 'a') as fileToWrite:
        fileToWrite.write(files.setStatisticsOff)
        fileToWrite.write(files.timeToShow)

# Join all queries in one file - Throughput Test - Oracle
def ReturnThroughputTestOracle():
    j = 0
    for x in files.ThroughputTest:
        if(j==files.sizeOfThroughput):
            break;
        j = j+1
        with open(files.destiny+"Throughput.sql", 'a') as fileToWrite:
            fileToWrite.write("-----------ThroughputTest-"+str(j)+"-begin-----------
\n\n")
        querynumber = 0
        for y in x:
            with open(files.path+str(y)+'.sql', 'r') as file:
                # read content from first file
                with open(files.destiny+"Throughput.sql", 'a') as fileToWrite:
                    if (j == 1 and y == 0):
                        for line in file:
                            # append content to second file
                            fileToWrite.write(line)
                        fileToWrite.write(files.setOracleStatisticsOn)
```

```
            elif (y != 0):
                if y in files.to_include:
                    fileToWrite.write(
                        "\n\n-----------Query "+str(y)+"-Begin-----------\n\n")
                    for line in file:
                        # append content to second file
                        fileToWrite.write(line)
                    fileToWrite.write(
                        "\n\n-----------Query "+str(y)+"-End-----------\n")
                elif not files.to_include:
                    fileToWrite.write(
                        "\n\n-----------Query "+str(y)+"-Begin-----------\n\n")
                    for line in file:
                        # append content to second file
                        fileToWrite.write(line)
                    fileToWrite.write(
                        "\n\n-----------Query "+str(y)+"-End-----------\n")
            else:
                continue
    with open(files.destiny+"Throughput.sql", 'a') as fileToWrite:
        fileToWrite.write(files.setOracleStatisticsOff)

# Power Test Oracle
def ReturnPowerTestOracle():
    j = 0


    for x in files.powertest:
        j = j+1
        querynumber = 0
        for y in x:
            if (y == 0 and j == 1):
```

```
            querynumber = 0
        else:
            querynumber = querynumber+1

        with open(files.path+str(y)+'.sql', 'r') as file:
            # read content from first file
            with open(files.destiny+"powertest.sql", 'a') as fileToWrite:
                if (y == 0 and j == 1):
                    for line in file:
                        # append content to second file
                        fileToWrite.write(line)
                    fileToWrite.write(files.setOracleStatisticsOn)
                elif (y != 0):
                    if y in files.to_include:
                        fileToWrite.write(
                            "\n\n-----------Query "+str(y)+"-Begin-----------\n\n")
                        for line in file:
                            # append content to second file
                            fileToWrite.write(line)
                        fileToWrite.write(
                            "\n\n-----------Query "+str(y)+"-End-----------\n")
                    elif not files.to_include:
                        fileToWrite.write(
                            "\n\n-----------Query "+str(y)+"-Begin-----------\n\n")
                        for line in file:
                            # append content to second file
                            fileToWrite.write(line)
                        fileToWrite.write(
                            "\n\n-----------Query "+str(y)+"-End-----------\n")

with open(files.destiny+"powertest.sql", 'a') as fileToWrite:
    fileToWrite.write(files.setOracleStatisticsOff)
```

APPENDIX B - PYTHON SCRIPT DATA EXTRACTION

```
import csv
import numpy
import files

# Join all queries in one file - Throughput Test
def ReturnThroughputTest():
    j = 0
    with open(files.destiny+"throughput", 'a') as fileToWrite:
        fileToWrite.write(files.timeToShow)
        fileToWrite.write(files.setStatisticsOn)
    for x in files.ThroughputTest:
        if(j==files.sizeOfThroughput):
            break;
        j = j+1
        with open(files.destiny+"throughput", 'a') as fileToWrite:
            fileToWrite.write(
                "print('-----------ThroughputTest-"+str(j)+"-begin-----------')\n\n")
        querynumber = 0
        for y in x:
            with open(files.path+str(y)+'.sql', 'r') as file:
                # read content from first file
                with open(files.destiny+"throughput", 'a') as fileToWrite:
                    if (j == 1 and y == 0):
                        for line in file:
                            # append content to second file
                            fileToWrite.write(line)
                    elif (y != 0):
                        if y in files.to_include:
                            fileToWrite.write(
                                "\n\nprint('-----------Query "+str(y)+"-Begin-----------')\n\n")
```

```
                    for line in file:
                        # append content to second file
                        fileToWrite.write(line)
                    fileToWrite.write(
                        "\n\nprint('-----------Query "+str(y)+"-End-----------')\n")
                elif not files.to_include:
                    fileToWrite.write(
                        "\n\nprint('-----------Query "+str(y)+"-Begin-----------')\n\n")
                    for line in file:
                        # append content to second file
                        fileToWrite.write(line)
                    fileToWrite.write(
                        "\n\nprint('-----------Query "+str(y)+"-End-----------')\n")
                else:
                    continue
        with open(files.destiny+"throughput", 'a') as fileToWrite:
            fileToWrite.write("\n\n\nprint('-----------ThroughputTest-"+str(j)+"        -
End-----------')\n\n\n")
    with open(files.destiny+"throughput", 'a') as fileToWrite:
        fileToWrite.write(files.setStatisticsOff)
        fileToWrite.write(files.timeToShow)

# Power Test SQL Server
def ReturnPowerTest():
    j = 0

    with open(files.destiny+files.fileName, 'a') as fileToWrite:
        fileToWrite.write(files.timeToShow)
        fileToWrite.write(files.setStatisticsOn)
        fileToWrite.write(
            "print('-----------Start of the power test-begin-----------')\n\n")
    for x in files.powertest:
```

```
        j = j+1
        with open(files.destiny+files.fileName, 'a') as fileToWrite:
            fileToWrite.write("print('-----------PowerTest-"       +str(j)+"-begin----------
')\n\n")
        querynumber = 0
        for y in x:
            if (y == 0 and j == 1):
                querynumber = 0
            else:
                querynumber = querynumber+1

            with open(files.path+str(y)+'.sql', 'r') as file:
                # read content from first file
                with open(files.destiny+files.fileName, 'a') as fileToWrite:
                    if (y == 0 and j == 1):
                        for line in file:
                            # append content to second file
                            fileToWrite.write(line)
                    elif (y != 0):
                        if y in files.to_include:
                            fileToWrite.write(
                                "\n\nprint('-----------Query "+str(y)+"-Begin----------')\n\n")
                            for line in file:
                                # append content to second file
                                fileToWrite.write(line)
                            fileToWrite.write(
                                "\n\nprint('-----------Query "+str(y)+"-End----------')\n")
                        elif not files.to_include:
                            fileToWrite.write(
                                "\n\nprint('-----------Query "+str(y)+"-Begin----------')\n\n")
                            for line in file:
                                # append content to second file
```

```
                    fileToWrite.write(line)
                fileToWrite.write(
                    "\n\nprint('-----------Query "+str(y)+"-End-----------')\n")

    with open(files.destiny+files.fileName, 'a') as fileToWrite:
        fileToWrite.write(
            "print('-----------PowerTest-End-----------')"+'\n\n\n')
        fileToWrite.write(files.setStatisticsOff)
        fileToWrite.write(files.timeToShow)


# Join all queries in one file - Throughput Test
def ReturnMySQLThroughputTest():
    j = 0
    for x in files.ThroughputTest:
        if(j==files.sizeOfThroughput):
            break;
        j = j+1
        with open(files.destiny+'ThroughputMySQLTest1-40.sql', 'a') as fileToWrite:
            fileToWrite.write(
                "-- ThroughputTest-"+str(j)+"-begin\n\n")
        querynumber = 0
        for y in x:
            with open(files.path+str(y)+'.sql', 'r') as file:
                # read content from first file
                with open(files.destiny+'ThroughputMySQLTest1-40.sql', 'a') as
fileToWrite:
                    if (j == 1 and y == 0):
                        for line in file:
                            # append content to second file
                            fileToWrite.write(line)
                    elif (y != 0):
                        if y in files.to_include:
```

```
                fileToWrite.write(
                    "\n\n-- Query "+str(y)+"-Begin\n\n")
                for line in file:
                    # append content to second file
                    fileToWrite.write(line)
                fileToWrite.write(
                    "\n\n-- Query "+str(y)+"-End\n")
            elif not files.to_include:
                fileToWrite.write(
                    "\n\n-- Query "+str(y)+"-Begin\n\n")
                for line in file:
                    # append content to second file
                    fileToWrite.write(line)
                fileToWrite.write(
                    "\n\n-- Query "+str(y)+"-End\n")
            else:
                continue
    with open(files.destiny+'ThroughputMySQLTest1-40.sql', 'a') as fileToWrite:
        fileToWrite.write("\n\n\n-- ThroughputTest-"+str(j)+" -End\n\n\n")
with open(files.destiny+'ThroughputMySQLTest1-40.sql', 'a') as fileToWrite:
    fileToWrite.write("\n\nDROP VIEW REVENUE0;\n")
    # fileToWrite.write(files.timeToShow)

# Power Test MySQL
def ReturnPowerTestMySQL():
    j = 0

    with open(files.destiny+'PowerTestMySQL.sql', 'a') as fileToWrite:
        # fileToWrite.write(files.timeToShow)
        # fileToWrite.write(files.setStatisticsOn)
        fileToWrite.write(
            "-- Start of the power test-begin\n\n")
```

```
for x in files.powertest:
    j = j+1
    with open(files.destiny+'PowerTestMySQL.sql', 'a') as fileToWrite:
        fileToWrite.write("-- PowerTest-" +str(j)+"-begin\n\n")
    querynumber = 0
    for y in x:
        if (y == 0 and j == 1):
            querynumber = 0
        else:
            querynumber = querynumber+1

        with open(files.path+str(y)+'.sql', 'r') as file:
            # read content from first file
            with open(files.destiny+'PowerTestMySQL.sql', 'a') as fileToWrite:
                if (y == 0 and j == 1):
                    for line in file:
                        # append content to second file
                        fileToWrite.write(line)
                elif (y != 0):
                    if y in files.to_include:
                        fileToWrite.write(
                            "\n\n-- Query "+str(y)+"-Begin)\n\n")
                        for line in file:
                            # append content to second file
                            fileToWrite.write(line)
                        fileToWrite.write(
                            "\n\n-- Query "+str(y)+"-End\n")
                    elif not files.to_include:
                        fileToWrite.write(
                            "\n\n-- Query "+str(y)+"-Begin\n\n")
                        for line in file:
                            # append content to second file
```

```
                fileToWrite.write(line)
            fileToWrite.write(
                "\n\n-- Query "+str(y)+"-End\n")

    with open(files.destiny+'PowerTestMySQL.sql', 'a') as fileToWrite:
        fileToWrite.write(
            "-- PowerTest-End"+'\n\n\n')
        # fileToWrite.write(files.setStatisticsOff)
        # fileToWrite.write(files.timeToShow)


#SQL Server way of converting throughput values returned by SQL Server console to excel
def ConvertValuesToExcel():
    path = files.pathEncrypted
    file = open(path+files.filenameTh, mode='r')

    # read all lines at once
    all_of_it = file.readlines()
    # Devo procurar as palavras chave em cada linha e tentar perceber que linha de execucao esta acontecer
    # adding header
    headerList = ['', 'Q1', 'Q2', 'Q3', 'Q4', 'Q5', 'Q6', 'Q7', 'Q8', 'Q9', 'Q10', 'Q11', 'Q12', 'Q13', 'Q14',
                  'Q15', 'Q16', 'Q17', 'Q18', 'Q19', 'Q20', 'Q21', 'Q22', 'Start Time', 'End Time', 'Execution Sum', 'Execution Sum Average']

    # converting data frame to csv
    # open CSV file and assign header

    with open(path+files.filenameCSVTh, 'w') as file:
        dw = csv.DictWriter(file, delimiter=',',
                            fieldnames=headerList, lineterminator='\n')
        dw.writeheader()
```

```
i = 0
data = []
countOfExecutionsNotBank = 0
while i < len(all_of_it):

    arrayElasedTime = ['', '', '', '', '', '', '', '', '', '', '',
                       '', '', '', '', '', '', '', '', '', '', '', '', '', '', '']
    arrayCPUTime = ['', '', '', '', '', '', '', '', '', '', '',
                    '', '', '', '', '', '', '', '', '', '', '', '', '', '', '']
    print('arrayCPUTime', len(arrayCPUTime))
    print('arrayElasedTime', len(arrayElasedTime))
    CPUBranch = ''
    ElapsedBranch = ''
    if 'throughputtest' in all_of_it[i].lower():
        # getting the execution index
        indexOnString = ''.join(
            [i for i in all_of_it[i] if i.isdigit()])
        print(all_of_it[i])
        CPUBranch = 't'+indexOnString+'-CPU'
        ElapsedBranch = 't'+indexOnString+'-ElapsedTime'
        arrayCPUTime[0] = CPUBranch
        arrayElasedTime[0] = ElapsedBranch
        if 'begin' in all_of_it[i].lower():
            # colocar na coluna de end time
            BeginTime = ""+all_of_it[i+1]+' '+all_of_it[i+2]+""
            BeginTime = BeginTime.replace("\n", " ")
            print('BeginTime', BeginTime)
            arrayElasedTime[23] = str(BeginTime)
    # else:
        j = i
        print(j)
        totalCPU = 0
```

```
totalElapsed = 0
# This is the place I am getting the times
while 'end' not in all_of_it[j].lower() and j < len(all_of_it)-1:
   # 1-get between executions the times from the queries
   if 'query' in all_of_it[j].lower():
      # getting the execution index
      indexOfQuery = ''.join(
         [i for i in all_of_it[j] if i.isdigit()])
      indexOfQuery = int(indexOfQuery)
      print('indexOfQuery', indexOfQuery)
      z = j
      while 'end' not in all_of_it[z].lower():
         # if 'cpu' in all_of_it[z].lower() or 'elapsed time' in
all_of_it[z].lower():
         if 'cpu' in all_of_it[z].lower() or 'elapsed time' in
all_of_it[z].lower():
            arrayCPU, arrayTime = all_of_it[z].split(
               ",", 1)
            arrayCPUTime[indexOfQuery] = ''.join(
               [i for i in arrayCPU if i.isdigit()]).replace("\n", " ")
            arrayElasedTime[indexOfQuery] = ''.join(
               [i for i in arrayTime if i.isdigit()]).replace("\n", " ")
            totalCPU = totalCPU + \
               int(arrayCPUTime[indexOfQuery])
            totalElapsed = totalElapsed + \
               int(arrayElasedTime[indexOfQuery])
            print('Query ', indexOfQuery)
            print('elapsed time ',
                  arrayElasedTime[indexOfQuery])
            print('CPU Time ', arrayCPUTime[indexOfQuery])
         z = z+1
      j = z
```

```
            j = j+1
        arrayCPUTime[26] = totalCPU/22
        arrayCPUTime[25] = totalCPU
        arrayElasedTime[26] = totalElapsed/22
        arrayElasedTime[25] = totalElapsed
        if (j+1 < len(all_of_it) or j+2 < len(all_of_it)):
            EndTime = ""+all_of_it[j+1]+' '+all_of_it[j+2]+""
        print('endtime', EndTime)
        print(all_of_it[j])
        arrayElasedTime[24] = EndTime
        i = j
        # data.append(arrayCPUTime)
        data.append(arrayElasedTime)
    i = i+1
SumTotalOfEachQueryForCPU, SumTotalOfEachQueryForElapsed, Num-
ber_of_Executions_Elapsed, Number_of_Executions_CPU = getTotalSum(
    data)
arrayElasedTime = ['', '', '', '', '', '', '', '', '', '', '',
                   '', '', '', '', '', '', '', '', '', '', '']
arrayCPUTime = ['', '', '', '', '', '', '', '', '', '', '',
                '', '', '', '', '', '', '', '', '', '', '']
AverageSumOfEachElapsedQuery = ['', '', '', '', '', '', '', '', '', '', '',
                                '', '', '', '', '', '', '', '', '', '', '']
AverageSumOfEachCPUQuery = ['', '', '', '', '', '', '', '', '', '', '',
                            '', '', '', '', '', '', '', '', '', '', '']

# AverageSumOfEachElapsedQuery = numpy.zeros(23)
# AverageSumOfEachCPUQuery = numpy.zeros(23)
arrayCPUTime[0] = 'CPU Total time'
AverageSumOfEachCPUQuery[0] = 'CPU Average time'
arrayElasedTime[0] = 'Elpased Total time'
AverageSumOfEachElapsedQuery[0] = 'Elpased Average time'
```

```
    resetindex = 0
    k = 0
    while k < len(SumTotalOfEachQueryForElapsed):
        if (SumTotalOfEachQueryForElapsed[k] != 0):
            countOfExecutionsNotBank = countOfExecutionsNotBank+1
        k = k+1
    k = 0


    while k < 22:
        arrayCPUTime[k+1] = SumTotalOfEachQueryForCPU[resetindex]
        arrayElasedTime[k+1] = SumTotalOfEachQueryForElapsed[resetindex]
        if (Number_of_Executions_Elapsed[resetindex+1] != 0):
            AverageSumOfEachElapsedQuery[k+1] = SumTotalOfEach-
QueryForElapsed[resetindex] / \
                Number_of_Executions_Elapsed[resetindex+1]
        else:
            AverageSumOfEachElapsedQuery[k+1] = 0
        if (Number_of_Executions_CPU[resetindex+1] != 0):
            AverageSumOfEachCPUQuery[k+1] = SumTotalOfEach-
QueryForCPU[resetindex] / \
                Number_of_Executions_CPU[resetindex+1]
        else:
            AverageSumOfEachCPUQuery[k+1] = 0
        resetindex = resetindex+1
        k = k+1
    data.append(arrayCPUTime)
    data.append(arrayElasedTime)
    data.append(AverageSumOfEachElapsedQuery)
    data.append(AverageSumOfEachCPUQuery)
    write = csv.writer(file, delimiter=",", lineterminator='\n')
    write.writerows(data)
```

```
#SQL Server way of converting powertest values returned by SQL Server console to excel
def ConvertValuesToExcelPowertest():

    path = files.pathEncryptedTDE
    file = open(path+files.filenamePt, mode='r')

    # read all lines at once
    all_of_it = file.readlines()
    # Devo procurar as palavras chave em cada linha e tentar perceber que linha de execucao esta acontecer
    # adding header
    headerList = ['', 'Q1', 'Q2', 'Q3', 'Q4', 'Q5', 'Q6', 'Q7', 'Q8', 'Q9', 'Q10', 'Q11', 'Q12', 'Q13', 'Q14',
                  'Q15', 'Q16', 'Q17', 'Q18', 'Q19', 'Q20', 'Q21', 'Q22', 'Start Time', 'End Time', 'Execution Sum', 'Execution Sum Average']

    # converting data frame to csv
    # open CSV file and assign header

    with open(path+files.filenameCSVPt, 'w') as file:
        dw = csv.DictWriter(file, delimiter=',',
                            fieldnames=headerList, lineterminator='\n')
        dw.writeheader()
        i = 0
        data = []
        countOfExecutionsNotBank = 0
        EndTime=0
        while i < len(all_of_it):

            arrayElasedTime = ['', '', '', '', '', '', '', '', '', '', '',
                               '', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '']
            arrayCPUTime = ['', '', '', '', '', '', '', '', '', '', '',
```

```
                ", ", ", ", ", ", ", ", ", ", ", ", ", ", ", ", ", ", "]
print('arrayCPUTime', len(arrayCPUTime))
print('arrayElasedTime', len(arrayElasedTime))
CPUBranch = ''
ElapsedBranch = ''
if 'powertest' in all_of_it[i].lower():
    # getting the execution index
    indexOnString = ''.join(
        [i for i in all_of_it[i] if i.isdigit()])
    print(all_of_it[i])
    CPUBranch = 't'+indexOnString+'-CPU'
    ElapsedBranch = 't'+indexOnString+'-ElapsedTime'
    arrayCPUTime[0] = CPUBranch
    arrayElasedTime[0] = ElapsedBranch
    if 'begin' in all_of_it[i].lower():
        # colocar na coluna de end time
        BeginTime = ""+all_of_it[i+1]+' '+all_of_it[i+2]+""
        BeginTime = BeginTime.replace("\n", " ")
        print('BeginTime', BeginTime)
        arrayElasedTime[23] = str(BeginTime)
# else:
    j = i
    print(j)
    totalCPU = 0
    totalElapsed = 0
    # Este e o sitio onde vou apanhar os tempos das queries
    while 'end' not in all_of_it[j].lower() and j < len(all_of_it)-1:
        # 1-get between executions the times from the queries
        if 'query' in all_of_it[j].lower():
            # getting the execution index
            indexOfQuery = ''.join(
                [i for i in all_of_it[j] if i.isdigit()])
```

```
                indexOfQuery = int(indexOfQuery)
                print('indexOfQuery', indexOfQuery)
                z = j
                while 'end' not in all_of_it[z].lower():
                  # if 'cpu' in all_of_it[z].lower() or 'elapsed time' in
all_of_it[z].lower():
                  if 'cpu' in all_of_it[z].lower() or 'elapsed time' in
all_of_it[z].lower():
                    arrayCPU, arrayTime = all_of_it[z].split(
                      ",", 1)
                    arrayCPUTime[indexOfQuery] = ''.join(
                      [i for i in arrayCPU if i.isdigit()]).replace("\n", " ")
                    arrayElasedTime[indexOfQuery] = ''.join(
                      [i for i in arrayTime if i.isdigit()]).replace("\n", " ")
                    totalCPU = totalCPU + \
                      int(arrayCPUTime[indexOfQuery])
                    totalElapsed = totalElapsed + \
                      int(arrayElasedTime[indexOfQuery])
                    print('Query ', indexOfQuery)
                    print('elapsed time ',
                        arrayElasedTime[indexOfQuery])
                    print('CPU Time ', arrayCPUTime[indexOfQuery])
                  z = z+1
                j = z
              j = j+1
            arrayCPUTime[26] = totalCPU/22
            arrayCPUTime[25] = totalCPU
            arrayElasedTime[26] = totalElapsed/22
            arrayElasedTime[25] = totalElapsed
            if (j+1 < len(all_of_it) or j+2 < len(all_of_it)):
              EndTime = ""+all_of_it[j+1]+' '+all_of_it[j+2]+""
            print(all_of_it[j])
```

```
            arrayElasedTime[24] = EndTime
            i = j
            data.append(arrayCPUTime)
            data.append(arrayElasedTime)
        i = i+1
    SumTotalOfEachQueryForCPU, SumTotalOfEachQueryForElapsed, Number_of_Executions_Elapsed, Number_of_Executions_CPU = getTotalSum(
        data)
    arrayElasedTime = ['', '', '', '', '', '', '', '', '', '', '',
                       '', '', '', '', '', '', '', '', '', '', '']
    arrayCPUTime = ['', '', '', '', '', '', '', '', '', '', '',
                    '', '', '', '', '', '', '', '', '', '', '']
    AverageSumOfEachElapsedQuery = ['', '', '', '', '', '', '', '', '', '', '',
                                    '', '', '', '', '', '', '', '', '', '', '']
    AverageSumOfEachCPUQuery = ['', '', '', '', '', '', '', '', '', '', '',
                                '', '', '', '', '', '', '', '', '', '', '']

    # AverageSumOfEachElapsedQuery = numpy.zeros(23)
    # AverageSumOfEachCPUQuery = numpy.zeros(23)
    arrayCPUTime[0] = 'CPU Total time'
    AverageSumOfEachCPUQuery[0] = 'CPU Average time'
    arrayElasedTime[0] = 'Elpased Total time'
    AverageSumOfEachElapsedQuery[0] = 'Elpased Average time'
    resetindex = 0
    k = 0
    while k < len(SumTotalOfEachQueryForElapsed):
        if (SumTotalOfEachQueryForElapsed[k] != 0):
            countOfExecutionsNotBank = countOfExecutionsNotBank+1
        k = k+1
    k = 0

    while k < 22:
```

```
            arrayCPUTime[k+1] = SumTotalOfEachQueryForCPU[resetindex]
            arrayElasedTime[k+1] = SumTotalOfEachQueryForElapsed[resetindex]
            if (Number_of_Executions_Elapsed[resetindex+1] != 0):
                AverageSumOfEachElapsedQuery[k+1] = SumTotalOfEach-
QueryForElapsed[resetindex] / \
                    Number_of_Executions_Elapsed[resetindex+1]
            else:
                AverageSumOfEachElapsedQuery[k+1] = 0
            if (Number_of_Executions_CPU[resetindex+1] != 0):
                AverageSumOfEachCPUQuery[k+1] = SumTotalOfEach-
QueryForCPU[resetindex] / \
                    Number_of_Executions_CPU[resetindex+1]
            else:
                AverageSumOfEachCPUQuery[k+1] = 0
            resetindex = resetindex+1
            k = k+1
        data.append(arrayCPUTime)
        data.append(arrayElasedTime)
        data.append(AverageSumOfEachElapsedQuery)
        data.append(AverageSumOfEachCPUQuery)
        write = csv.writer(file, delimiter=",", lineterminator='\n')
        write.writerows(data)


#MySQL way of converting powertest values returned by MySQL console to excel
def ConvertValuesMySQLToExcelPowertestTeeVersion():
    file_path = files.pathEncrypted
    headerList = ['', 'Q1', 'Q2', 'Q3', 'Q4', 'Q5', 'Q6', 'Q7', 'Q8', 'Q9', 'Q10', 'Q11',
'Q12', 'Q13', 'Q14','Q15', 'Q16', 'Q17', 'Q18', 'Q19', 'Q20', 'Q21', 'Q22', 'Execution
Sum', 'Execution Sum Average']
    data = []
    queries_order = [0, 14, 2, 9, 20, 6, 17, 18, 8, 21, 13, 3, 22, 16, 4, 11, 15, 1, 10, 19,
5, 7, 12]
    k=0
```

```
    # Initialize arrays
    arrayElapsedTime = [''] * 27
    with open(file_path + files.filenameCSVPt, 'w', newline='') as wfile:
        dw = csv.DictWriter(wfile, delimiter=',', fieldnames=headerList)
        dw.writeheader()
        with open(file_path + files.filenamePt, 'r') as file:
            lines = file.readlines()
            i=0
            for j,line in enumerate(lines):
                if 'row in set' in line or 'rows in set' in line or 'Empty set' in line:
                    # if k==0 or i == k or i==22:
                    if i==22:
                        k=k+1
                        continue
                    # Extracting the duration time
                    duration_str = line.split('(')[-1].split(')')[0]
                    duration_seconds = parse_duration_to_seconds(duration_str)
                    arrayElapsedTime[queries_order[i + 1]] = duration_seconds*1000
                    print(i)
                    print(duration_seconds)
                    k=k+1
                    i=i+1


            # Calculate total and average Elapsed time
            total_elapsed_time = sum(float(time) for time in arrayElapsedTime if time
!= '')
            average_elapsed_time = total_elapsed_time / len([time for time in ar-
rayElapsedTime if time != ''])


            arrayElapsedTime[0] = 'Elapsed Total time'
            arrayElapsedTime[23] = total_elapsed_time
            arrayElapsedTime[24] = average_elapsed_time
```

```
            data.append(arrayElapsedTime)

        write = csv.writer(wfile, delimiter=",")
        write.writerows(data)

    return data
#MySQL way of converting throughput values returned by MySQL console to excel

def ConvertValuesMySQLToExcelThroughputTeeVersion():
    file_path = files.pathEncrypted
    headerList = ['', 'Q1', 'Q2', 'Q3', 'Q4', 'Q5', 'Q6', 'Q7', 'Q8', 'Q9', 'Q10', 'Q11',
                  'Q12', 'Q13', 'Q14', 'Q15', 'Q16', 'Q17', 'Q18', 'Q19', 'Q20', 'Q21', 'Q22',
                  'Execution Sum', 'Execution Sum Average']
    data = []

    # Use the throughput order from files.ThroughputTest (excluding the first index in each sublist)
    throughput_order = [sublist[1:] for sublist in files.ThroughputTest[:files.sizeOfThroughput]]

    with open(file_path + files.filenameCSVTh, 'w', newline='') as wfile:
        dw = csv.DictWriter(wfile, delimiter=',', fieldnames=headerList)
        dw.writeheader()

        with open(file_path + files.filenameTh, 'r') as file:
            lines = file.readlines()
            line_index = 0

            for execution_order in throughput_order:
                execution_times = [''] * 27
                k = 0
```

```
            while line_index < len(lines) and k < len(execution_order):
                line = lines[line_index]
                if ('row in set' in line or 'rows in set' in line or 'Empty set' in line):
                    duration_str = line.split('(')[-1].split(')')[0]
                    duration_seconds = parse_duration_to_seconds(duration_str)
                    query_position = execution_order[k]
                    execution_times[query_position] = duration_seconds * 1000  # Convert to ms
                    k += 1
                line_index += 1

            total_elapsed_time = sum(t for t in execution_times if isinstance(t, float))
            average_elapsed_time = total_elapsed_time / len([t for t in execution_times if isinstance(t, float)])

            execution_times[0] = 'Elapsed Total time'
            execution_times[23] = total_elapsed_time
            execution_times[24] = average_elapsed_time

            data.append(execution_times[:])

        writer = csv.writer(wfile, delimiter=",")
        writer.writerows(data)

    return data
def ConvertValuesOracleToExcelPowertest():
    file_path = files.pathEncrypted  # Adjust this path as necessary
    headerList = ['', 'Q1', 'Q2', 'Q3', 'Q4', 'Q5', 'Q6', 'Q7', 'Q8', 'Q9', 'Q10', 'Q11', 'Q12', 'Q13', 'Q14','Q15', 'Q16', 'Q17', 'Q18', 'Q19', 'Q20', 'Q21', 'Q22', 'Execution Sum (ms)', 'Execution Sum Average (ms)']
    data = []
```

```
    queries_order = [0, 14, 2, 9, 20, 6, 17, 18, 8, 21, 13, 3, 22, 16, 4, 11, 15, 1, 10, 19, 5, 7, 12]


    # Initialize arrays
    arrayElapsedTime = [''] * 27
    with open(file_path + files.filenameCSVPt, 'w') as wfile:
        dw = csv.DictWriter(wfile, delimiter=',', fieldnames=headerList, lineterminator='\n')
        dw.writeheader()


        with open(file_path + files.filenamePt, 'r') as file:
            lines = file.readlines()
            k = 1
            for line in lines:
                # Match 'Elapsed: HH:MM:SS.SS' and extract the time
                match = re.search(r'Elapsed:\s*(\d{2}:\d{2}:\d{2}\.\d{2})', line)
                if match and k < len(queries_order):
                    elapsed_time_str = match.group(1)
                    # Convert elapsed time toms
                    hours, minutes, seconds = map(float, elapsed_time_str.split(':'))
                    elapsed_time_ms = (hours * 3600 + minutes * 60 + seconds) * 1000 # convert toms


                    # Store the elapsed time inms in the array at the correct position
                    arrayElapsedTime[queries_order[k]] = elapsed_time_ms
                    k += 1


            # Calculate total and average Elapsed time inms
            total_elapsed_time_ms = sum(time for time in arrayElapsedTime if time != '')
            average_elapsed_time_ms = total_elapsed_time_ms / len([time for time in arrayElapsedTime if time != ''])
```

```
            arrayElapsedTime[0] = 'Elapsed Total time (ms)'
            arrayElapsedTime[23] = total_elapsed_time_ms
            arrayElapsedTime[24] = average_elapsed_time_ms

            data.append(arrayElapsedTime)
            writer = csv.writer(wfile, delimiter=",", lineterminator='\n')
            writer.writerows(data)
    return data

#function to be used to get the total sum of the
def getTotalSum(data):
    QuerySumElapsed = numpy.zeros(22)
    QuerySumCPU = numpy.zeros(22)
    Number_of_Executions_Elapsed = numpy.zeros(23)
    Number_of_Executions_CPU = numpy.zeros(23)
    for x in data:
        i = 0
        j = 0
        if ('cpu' in x[0].lower()):
            while i < 23:
                if (i > 0 and i <= 23 and x[i] != ''):
                    QuerySumCPU[j] = QuerySumCPU[j]+float(x[i])
                    Number_of_Executions_CPU[i] = Number_of_Executions_CPU[i]+1
                    j = j+1
                if (x[i] == '' and i > 0 and i <= 23):
                    QuerySumCPU[j] = QuerySumCPU[j]+0
                    j = j+1
                i = i+1
        elif ('ElapsedTime' in x[0] or 'Elapsed Total time' in x[0] or isinstance(x[0], (int, float, complex))):
            while i < 23:
```

```
            if (i > 0 and i <= 23 and x[i] != ''):
                QuerySumElapsed[j] = QuerySumElapsed[j]+float(x[i])
                Number_of_Executions_Elapsed[i] = Number_of_Executions_Elapsed[i]+1
                j = j+1
            if (x[i] == '' and i > 0 and i <= 23):
                QuerySumElapsed[j] = QuerySumElapsed[j]+0
                j = j+1
            i = i+1
    return QuerySumCPU, QuerySumElapsed, Number_of_Executions_Elapsed, Number_of_Executions_CPU


#it converts string to seconds
def parse_duration_to_seconds(duration_str):
    """
    Convert duration string to seconds.
    Supports formats like "1 min 35.05 sec" or "1 hour 2 min 35 sec".
    """
    # Splitting the duration string by space
    duration_parts = duration_str.split()
# Initializing variables to store minutes and seconds
    minutes = 0
    seconds = 0
# Iterating over each part of the duration string
    for i in range(len(duration_parts)):
        if duration_parts[i] == 'min':
            # Converting the part before 'min' to minutes
            minutes += float(duration_parts[i - 1])
        elif duration_parts[i] == 'sec':
            # Converting the part before 'sec' to seconds
            seconds += float(duration_parts[i - 1])
    # Converting minutes to seconds and adding to the total seconds
```

```
    seconds += minutes * 60
    return seconds
```

APPENDIX C – AUXILIARY FUNCTIONS FOR PYTHON MULTI-QUERY EXECUTION MANAGEMENT AND DATA EXTRACTION SCRIPTS

```
# 0 in the array means the file which has the create views or other code that needs to be executed at the beginning of the batch
powertest = [[0, 14, 2, 9, 20, 6, 17, 18, 8, 21,
            13, 3, 22, 16, 4, 11, 15, 1, 10, 19, 5, 7, 12]]
to_include = []
ThroughputTest = [
    [0, 21, 3, 18, 5, 11, 7, 6, 20, 17, 12, 16,
        15, 13, 10, 2, 8, 14, 19, 9, 22, 1, 4],
    [0, 6, 17, 14, 16, 19, 10, 9, 2, 15, 8, 5,
        22, 12, 7, 13, 18, 1, 4, 20, 3, 11, 21],
    [0, 8, 5, 4, 6, 17, 7, 1, 18, 22, 14, 9, 10,
        15, 11, 20, 2, 21, 19, 13, 16, 12, 3],
    [0, 5, 21, 14, 19, 15, 17, 12, 6, 4, 9, 8,
        16, 11, 2, 10, 18, 1, 13, 7, 22, 3, 20],
    [0, 21, 15, 4, 6, 7, 16, 19, 18, 14, 22, 11,
        13, 3, 1, 2, 5, 8, 20, 12, 17, 10, 9],
    [0, 10, 3, 15, 13, 6, 8, 9, 7, 4, 11, 22,
        18, 12, 1, 5, 16, 2, 14, 19, 20, 17, 21],
    [0, 18, 8, 20, 21, 2, 4, 22, 17, 1, 11, 9,
        19, 3, 13, 5, 7, 10, 16, 6, 14, 15, 12],
    [0, 19, 1, 15, 17, 5, 8, 9, 12, 14, 7, 4, 3,
        20, 16, 6, 22, 10, 13, 2, 21, 18, 11],
    [0, 8, 13, 2, 20, 17, 3, 6, 21, 18, 11, 19,
        10, 15, 4, 22, 1, 7, 12, 9, 14, 5, 16],
```

[0, 6, 15, 18, 17, 12, 1, 7, 2, 22, 13, 21,
10, 14, 9, 3, 16, 20, 19, 11, 4, 8, 5],
[0, 15, 14, 18, 17, 10, 20, 16, 11, 1, 8, 4,
22, 5, 12, 3, 9, 21, 2, 13, 6, 19, 7],
[0, 1, 7, 16, 17, 18, 22, 12, 6, 8, 9, 11,
4, 2, 5, 20, 21, 13, 10, 19, 3, 14, 15],
[0, 21, 17, 7, 3, 1, 10, 12, 22, 9, 16, 6,
11, 2, 4, 5, 14, 8, 20, 13, 18, 15, 19],
[0, 2, 9, 5, 4, 18, 1, 20, 15, 16, 17, 7,
21, 13, 14, 19, 8, 22, 11, 10, 3, 12, 6],
[0, 16, 9, 17, 8, 14, 11, 10, 12, 6, 21, 7,
3, 15, 5, 22, 20, 1, 13, 19, 2, 4, 18],
[0, 1, 3, 6, 5, 2, 16, 14, 22, 17, 20, 4, 9,
10, 11, 15, 8, 12, 19, 18, 13, 7, 21],
[0, 3, 16, 5, 11, 21, 9, 2, 15, 10, 18, 17,
7, 8, 19, 14, 13, 1, 4, 22, 20, 6, 12],
[0, 14, 4, 13, 5, 21, 11, 8, 6, 3, 17, 2,
20, 1, 19, 10, 9, 12, 18, 15, 7, 22, 16],
[0, 4, 12, 22, 14, 5, 15, 16, 2, 8, 10, 17,
9, 21, 7, 3, 6, 13, 18, 11, 20, 19, 1],
[0, 16, 15, 14, 13, 4, 22, 18, 19, 7, 1, 12,
17, 5, 10, 20, 3, 9, 21, 11, 2, 6, 8],
[0, 20, 14, 21, 12, 15, 17, 4, 19, 13, 10,
11, 1, 16, 5, 18, 7, 8, 22, 9, 6, 3, 2],
[0, 16, 14, 13, 2, 21, 10, 11, 4, 1, 22, 18,
12, 19, 5, 7, 8, 6, 3, 15, 20, 9, 17],
[0, 18, 15, 9, 14, 12, 2, 8, 11, 22, 21, 16,
1, 6, 17, 5, 10, 19, 4, 20, 13, 3, 7],
[0, 7, 3, 10, 14, 13, 21, 18, 6, 20, 4, 9,
8, 22, 15, 2, 1, 5, 12, 19, 17, 11, 16],
[0, 18, 1, 13, 7, 16, 10, 14, 2, 19, 5, 21,
11, 22, 15, 8, 17, 20, 3, 4, 12, 6, 9],

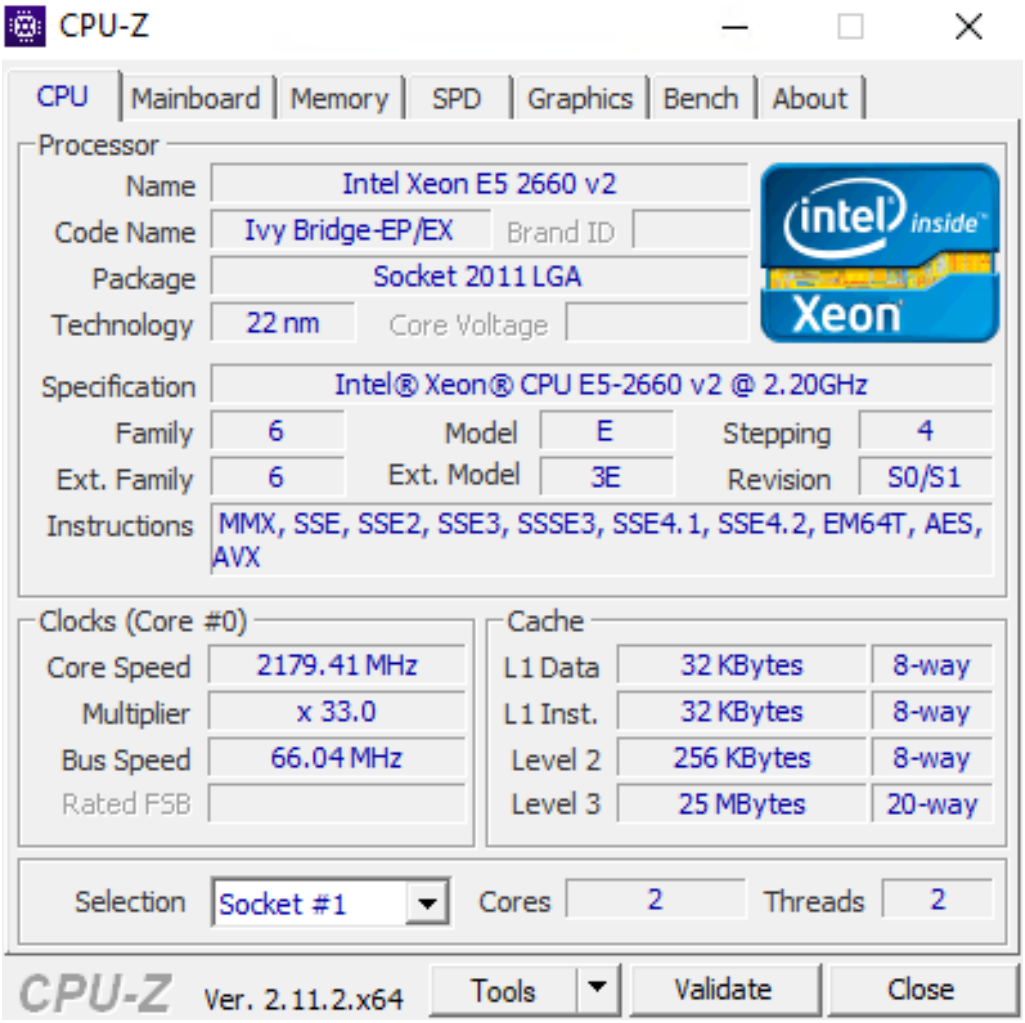
```
[0, 13, 2, 22, 5, 11, 21, 20, 14, 7, 10, 4,
   9, 19, 18, 6, 3, 1, 8, 15, 12, 17, 16],
[0, 14, 17, 21, 8, 2, 9, 6, 4, 5, 13, 22, 7,
   15, 3, 1, 18, 16, 11, 10, 12, 20, 19],
[0, 10, 22, 1, 12, 13, 18, 21, 20, 2, 14,
   16, 7, 15, 3, 4, 17, 5, 19, 6, 8, 9, 11],
[0, 10, 8, 9, 18, 12, 6, 1, 5, 20, 11, 17,
   22, 16, 3, 13, 2, 15, 21, 14, 19, 7, 4],
[0, 7, 17, 22, 5, 3, 10, 13, 18, 9, 1, 14,
   15, 21, 19, 16, 12, 8, 6, 11, 20, 4, 2],
[0, 2, 9, 21, 3, 4, 7, 1, 11, 16, 5, 20, 19,
   18, 8, 17, 13, 10, 12, 15, 6, 14, 22],
[0, 15, 12, 8, 4, 22, 13, 16, 17, 18, 3, 7,
   5, 6, 1, 9, 11, 21, 10, 14, 20, 19, 2],
[0, 15, 16, 2, 11, 17, 7, 5, 14, 20, 4, 21,
   3, 10, 9, 12, 8, 13, 6, 18, 19, 22, 1],
[0, 1, 13, 11, 3, 4, 21, 6, 14, 15, 22, 18,
   9, 7, 5, 10, 20, 12, 16, 17, 8, 19, 2],
[0, 14, 17, 22, 20, 8, 16, 5, 10, 1, 13, 2,
   21, 12, 9, 4, 18, 3, 7, 6, 19, 15, 11],
[0, 9, 17, 7, 4, 5, 13, 21, 18, 11, 3, 22,
   1, 6, 16, 20, 14, 15, 10, 8, 2, 12, 19],
[0, 13, 14, 5, 22, 19, 11, 9, 6, 18, 15, 8,
   10, 7, 4, 17, 16, 3, 1, 12, 2, 21, 20],
[0, 20, 5, 4, 14, 11, 1, 6, 16, 8, 22, 7, 3,
   2, 12, 21, 19, 17, 13, 10, 15, 18, 9],
[0, 3, 7, 14, 15, 6, 5, 21, 20, 18, 10, 4,
   16, 19, 1, 13, 9, 8, 17, 11, 12, 22, 2],
[0, 13, 15, 17, 1, 22, 11, 3, 4, 7, 20, 14, 21, 9, 8, 2, 18, 16, 6, 10, 12, 5, 19]]

# how many executions in a script
sizeOfThroughput=2
```

```
# ----------------Variables----------------
# Return Powertest and ThroughputTest
# from where the file is reading
fromFolder = ''
#to where the file is writing
toFolder = ''

# to where are the scripts heading to
path = fromFolder
destiny = toFolder
# used to five a name to the file when creating scripts to execute the queries
fileName='throughput.sql'

# Convert To Excel - Thios paths are the ones to be used when you want to convert a txt file to excel
pathEncrypted = ''
# file names to get the name of the file we are reading and the to give the name of the file we are saving
filenameTh = 'throughput.txt'
filenameCSVTh = "throughput.csv"
filenamePt = 'powertest.txt'
filenameCSVPt = "powertest.csv"
# Generic Variables
timeToShow = "\nprint(CONVERT (DATE, CURRENT_TIMESTAMP))print(CONVERT (TIME, CURRENT_TIMESTAMP))\n\n"
setStatisticsOn = "SET STATISTICS TIME ON;\n"
setStatisticsOff = "SET STATISTICS TIME OFF;"
setOracleStatisticsOn="-- Print the current timestamp \nset timing on \nSPOOL C:/Users/user/Desktop/oracleresults/execution_times.log \nSET SERVEROUTPUT ON \nSET HEADING OFF \nSET FEEDBACK OFF \n-- Output the current date and time to the spool file using PL/SQL \nBEGIN \n DBMS_OUTPUT.PUT_LINE('Script Start Time: ' || TO_CHAR(SYSDATE, 'MM/DD/YYYY HH24:MI:SS')); \nEND;\n/ \nSET HEADING ON \nSET AUTOTRACE TRACEONLY STATISTICS \nSET FEEDBACK OFF"
```

setOracleStatisticsOff="-- Stop spooling to end logging \nBEGIN\n DBMS_OUTPUT.PUT_LINE('Script Start Time: ' || TO_CHAR(SYSDATE, 'MM/DD/YYYY HH24:MI:SS'));\nEND;\n/\nSPOOL OFF\nSET TERMOUT ON\nDROP VIEW revenue0;"

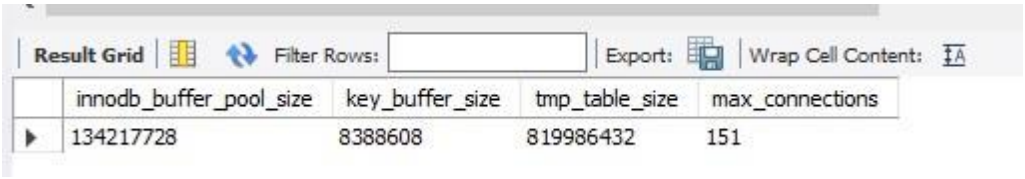
----------Variables----------

APPENDIX D – CPU-Z INFORMATION FOR AES-NI SUPPORT TO CPU



APPENDIX E – RDBMS SETUP PARAMETERS USED IN THE EXPERIMENTAL ENVIRONMENT

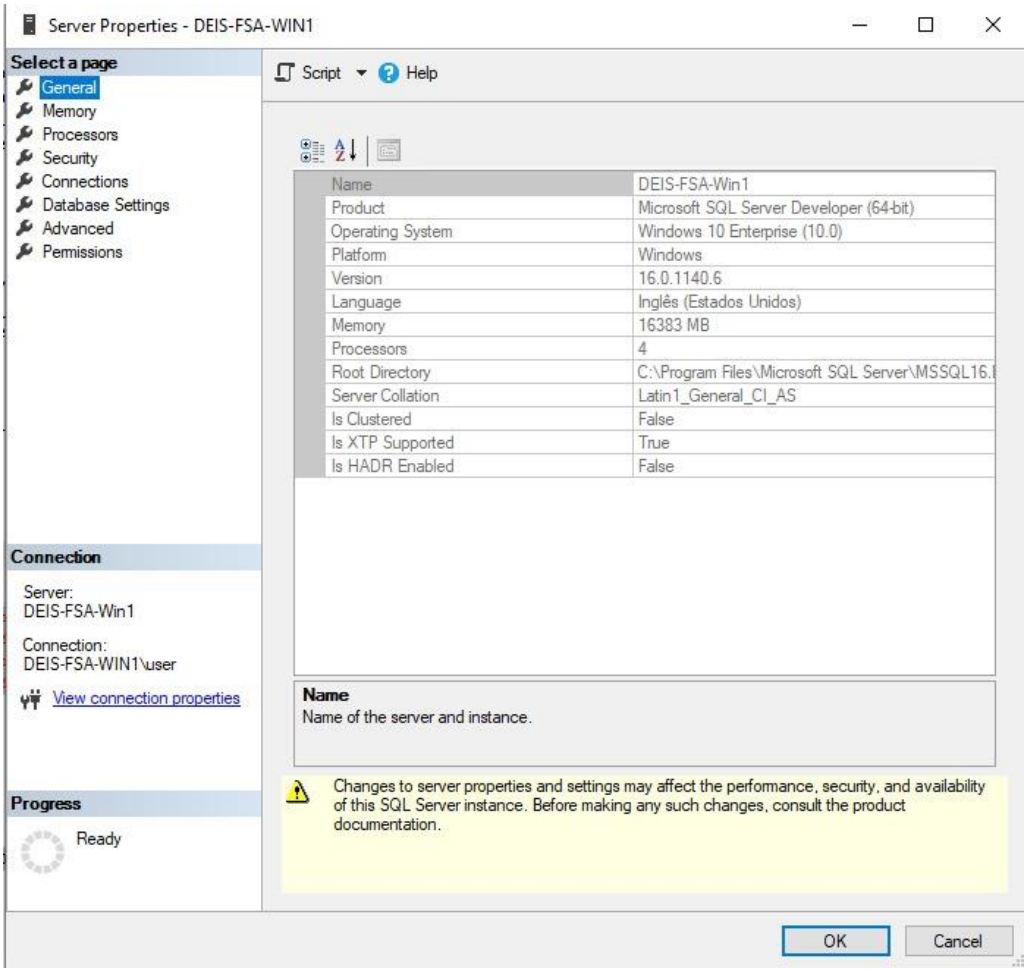
MySQL

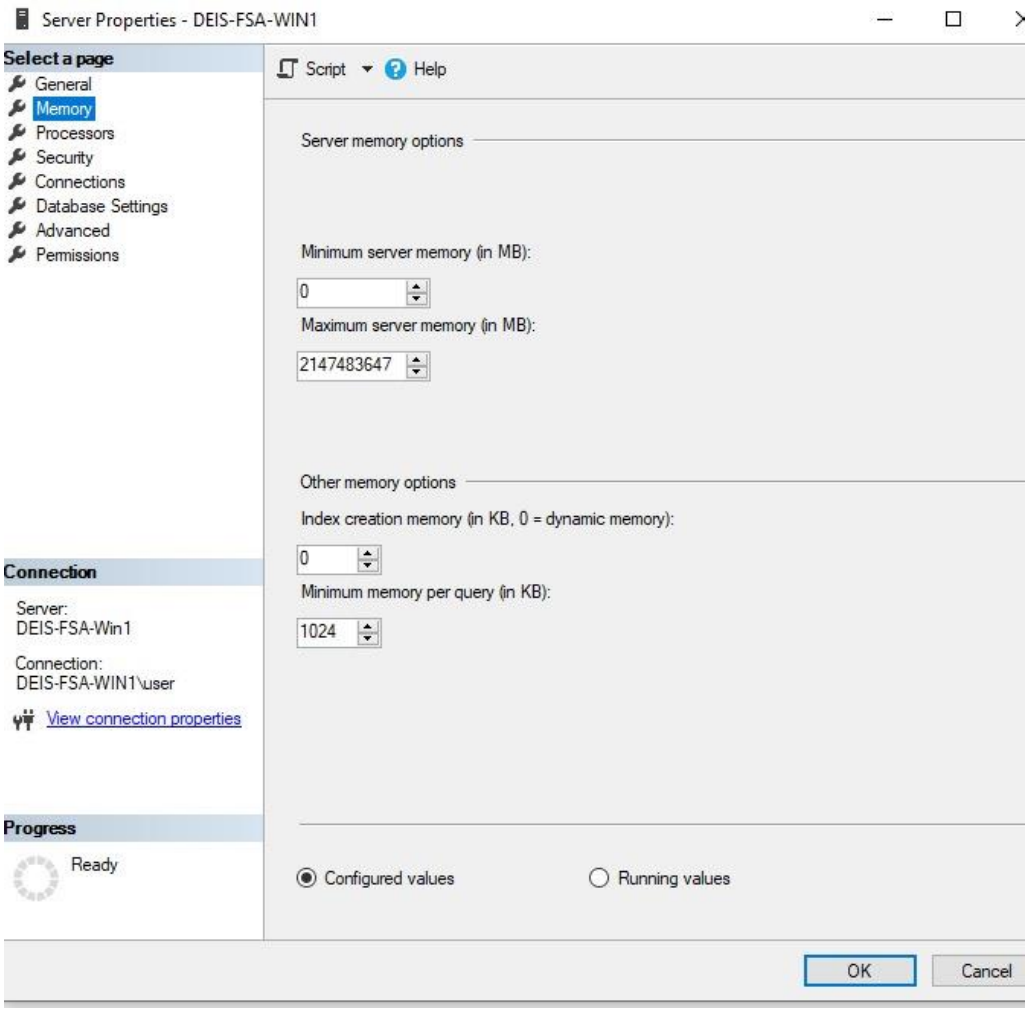


Result Grid | Filter Rows: | Export: | Wrap Cell Content:

innodb_buffer_pool_size	key_buffer_size	tmp_table_size	max_connections
134217728	8388608	819986432	151

SQL Server





Oracle

```
SQL> SHOW PARAMETER sga;

NAME                                 TYPE        VALUE
------------------------------------ ----------- ------------------------------
allow_group_access_to_sga            boolean     FALSE
lock_sga                             boolean     FALSE
pre_page_sga                         boolean     TRUE
sga_max_size                         big integer 4928M
sga_min_size                         big integer 0
sga_target                           big integer 4928M
unified_audit_sga_queue_size         integer     1048576
SQL> SHOW PARAMETER pga;

NAME                                 TYPE        VALUE
------------------------------------ ----------- ------------------------------
pga_aggregate_limit                  big integer 3278M
pga_aggregate_target                 big integer 1639M
SQL>
```

Instituto Superior de Engenharia
Politécnico de Coimbra