Université Jean Monnet

Centre INRAE Clermont-Auvergne-Rhône-Alpes

École des Mines de Saint-Étienne

M2 Master Internship Report

April 2024 - August 2024

Implementing an Energy-Efficient Database: A Methodological Approach

Submitted by

Sivaratnam PACHAVA

Cyber-Physical Social Systems (CPS2)

Internship Supervisor

François PINET

Directeur de Recherche, UR TSCF, Clermont-Ferrand, INRAE

David SARRAMIA

Maître de conférences, Laboratoire de Physique de Clermont-Auvergne

Myoung-Ah KANG

Maître de conférences, ISIMA, Université Clermont-Auvergne

Academic Supervisor

Pierre Maret

Professor in Computer Science, Université Jean Monnet, Saint-Etienne

Abstract

Data centers are essential infrastructures that consume significant amounts of energy to operate large-scale internet-based services. Recently, Increasing energy costs have become one of the critical issues in database centers. Energy consumption models are essential for designing and optimizing energy-efficient operations to reduce excessive energy use. However, databases currently lack the ability to manage energy consumption during operation, with past research primarily focusing on performance improvements. Several initiatives have been proposed to address this issue. In this paper, we survey the state-of-the-art techniques used for energy consumption modeling and prediction for data centers and their components. By studying existing literature on data center power modeling techniques, we develop a methodology to reduce energy consumption in databases. In physical design, where optimization structures (indexes, materialized views) are selected to satisfy query performance for a given workload, we develop an algorithm to select an efficient index for the given workload, and we analyze both query performance and energy consumption of the database by using selected indexes and without using indexes. We measured total energy consumption at the server level, without needing to measure individual hardware components. We have implemented our methodology and conducted experiments with different workloads to measure how efficiently indexes can optimize energy consumption in databases.

Contents

[1] Work Repository: https://github.com/sivapachava/INRAE_M2_Stage_Work_April_August_2024

List of Figures

1 Introduction

Developers of database management systems have traditionally focused their optimization efforts on minimizing either response time or resource usage required to run queries. From this perspective, the priority has been to maximize the performance of database applications by reducing time, thereby increasing system throughput. This trend is also present in other computing domains such as parallel processing, mobile computing, and real-time systems. Researchers, developers, and administrators need to understand how their design choices and policies affect the energy cost of the entire system and its various components.

The criticality of data centers has been fueled mainly by two phenomena. First, the ever-increasing growth in the demand for data computing, processing, and storage by a variety of large-scale cloud services [1], such as Google and Facebook, by telecommunication operators like British Telecom [2], by banks and others, has resulted in the proliferation of large data centers with thousands or even millions of servers. Second, the need to support a vast variety of applications, ranging from those that run for a few seconds to those that run persistently on shared hardware platforms, has promoted the building of large-scale computing infrastructures. As a result, data centers are considered one of the key enabling technologies for the fast-growing IT industry. However, these large-scale computing infrastructures have huge energy budgets, leading to various energy efficiency issues.

Moreover, energy efficiency in data centers has become crucial in recent years due to its (i) high economic, (ii) environmental, and (iii) performance impacts. Economically, a typical data center may consume as much energy as 25,000 households and can use 100 to 200 times as much electricity as standard office spaces. Furthermore, the energy costs of powering a typical data center double every five years, making power bills a significant expense, sometimes exceeding the cost of purchasing hardware [3]. Environmentally, data center energy usage contributes to substantial problems, prompting energy efficiency to become a chief concern for data center operators, ahead of traditional considerations like availability and security. Performance-wise, even idle servers consume a significant amount of energy [4]. Large savings can be made by turning off these servers and employing workload consolidation. However, these power-saving techniques can reduce system performance, necessitating a complex balance between energy savings and high performance.

With the emergence of the big data age and data-centric computing trends, the substantial amount of energy consumed by database systems has become a major concern in the pursuit of green information technology. The energy consumed by large-scale data management systems can be broadly categorized into two parts: energy used by IT equipment (servers, networks, storage, etc.) and energy used by infrastructure facilities (cooling and power conditioning systems). The amount of energy consumed by these components depends on the design of the data center and the efficiency of the equipment. For example, according to statistics published by [5], [6], the largest energy consumption in a typical data center is half of the total energy (50%) particularly consumed by IT equipment (CPU, memory, disk, etc.), and cooling infrastructure (35%) rank second in energy consumption (shown in Fig. 1).

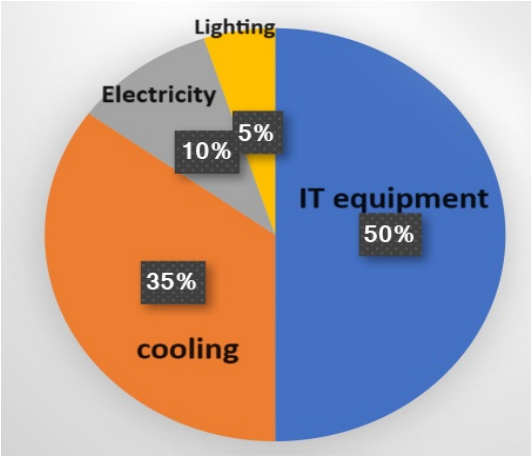


Figure 1: Energy consumption for different components in a data center

The database community has made no significant efforts to propose initiatives to reduce energy consumption. Traditionally, designing a database application focused on improving low-latency query pro-

cessing to satisfy the needs of end users. This focus has expanded to include the energy dimension during the deployment and exploitation phases of databases. For the past decade, research has focused on developing energy-efficient database systems to reduce economic costs and environmental impact. More and more researchers are proposing to build energy-efficient database systems to reduce energy consumption and improve energy efficiency. Consequently, energy management of database systems has emerged as an important research topic over the past few years. Compared with traditional performance-driven database designs, the main optimization and design goal of energy-efficient database systems is to minimize energy costs without sacrificing scalability while maintaining acceptable performance levels required by users.

However, reducing power consumption alone may not be reached to significant energy savings if the server continues running for the same amount of time and performing the same workload. To achieve meaningful energy savings, power reduction strategies must be combined with optimizations that decrease the overall workload or increase processing efficiency. Therefore, while reducing power consumption in databases is a step towards reducing energy consumption, broader optimizations and strategies are essential for significant energy savings. Various solutions have been proposed to address energy efficiency, considering both software and hardware aspects. On the software side, some studies have focused on energy-aware query processing. These studies follow two main approaches: (i) the definition of cost models to predict energy and (ii) the proposition of cost-driven techniques for reducing energy [7]. These cost models are integral to query optimizers, helping to estimate the execution costs of different query plans. However, these efforts are insufficient for advanced databases that handle large amounts of data and complex queries involving multiple tables, indicating the need for further research. On the hardware side, simple techniques like disabling electronic components during inactivity have been proposed, along with more advanced methods such as dynamically adjusting the performance of hardware components for better energy efficiency, like Dynamic Voltage and Frequency Scaling (DVFS) [8]. Our initiative concerns the query processing by creating indexes at the physical design phase of the database.

The remainder of this report is organized as follows: Section 2 highlights the objectives of this internship work, Section 3 provides a broad overview of the background and basic concepts related to energy optimization techniques, Section 4 reviews various related works aimed at improving energy efficiency in database environments, Section 5 details the proposed methodology based on the studies described in Section 4. Section 6 discusses the implementation process of our methodology, Section 7 presents the results of performance and energy consumption for database workloads, Section 8 offers the conclusion of our work, and Section 9 outlines future work for the next two months of the internship and suggests possible improvements to the current implementation.

2 Objectives of the Internship

- **Study existing methods:** Read, analyze, and review existing methods that present energy models for both SQL and NoSQL databases. Focus on optimizations that account for energy consumption alongside traditional performance metrics.
- **Propose and implement an optimization method:** Based on the initial study, develop a new optimization method that integrates energy consumption considerations. Implement this optimization method, exploring potential areas for improvement.
- **Evaluate the optimization method:** Test the applicability and effectiveness of the proposed optimization method on the Environmental Cloud for the Benefit of Auvergne (CEBA) platform specifically using PostgreSQL and Elasticsearch.

3 Presentation of the Context

This section discusses the context of my internship, beginning with an introduction to the host organization, INRAE. I am doing my 2nd-year Master's internship at the National Research Institute for Agriculture, Food, and the Environment(INRAE)[1]. This public research organization was formed in 2020 from the merger of the National Institute of Agronomic Research (INRA) and the National Institute of Research in Science and Technology for Environment and Agriculture (IRSTEA). INRAE focuses on research in agriculture, food, and the environment, addressing issues related to energy, chemistry, health, and territories.

[1] https://www.inrae.fr/

INRAE comprises 14 scientific departments spread across 18 regional centers in mainland France and overseas territories, with headquarters in the Paris region. Overall, INRAE consists of 268 research, service, and experimental units.

My internship took place at the INRAE Center in Clermont Auvergne-Rhône-Alpes[2], which includes the units in the Auvergne region.This center encompasses 14 research units and 14 experimental systems, employing a total of 840 agents (both permanent and contractual).

I worked in the "Technologies et systèmes d'information pour les agrosystèmes - Clermont-Ferrand (TSCF)[3]" unit, which focuses on engineering sciences and information and communication technologies to develop methods and tools for agro-environmental system engineering. The TSCF unit, consisting of 60 agents, and this unit conducts research, expertise, and testing in agricultural equipment security and performance, aiming to enhance agricultural safety and reduce pollution from agricultural activities. Through its technological research, the unit addresses the needs of productive, ecologically responsible agriculture and environmental management.

3.1 Background

3.1.1 Energy

Energy (E) represents the total amount of work done by a system during a specified time period (T), whereas power (P) indicates the rate at which this work is performed. The relationship between these quantities is mathematically described as E = PT, where E denotes the system's energy consumption measured in Joules, P is measured in Watts, and T represents the time in seconds.

To effectively analyze and manage a data center's energy consumption, the first step is to measure the energy usage of its components and determine where the majority of the energy is being utilized. By utilizing selected input features such as hardware, the DBMS, and query workload, an energy consumption model is constructed using techniques like regression and machine learning. A key challenge in this step is that certain critical system parameters, like the power consumption of specific components, cannot be directly measured. The outcome of this step is the creation of both a power model and an execution time model.

Once the models are developed, they need to be validated using training data workloads to ensure their accuracy and efficiency. Upon validation, these models can predict the energy consumption of components or the entire system. These predictions are crucial for enhancing the energy efficiency of the data center. By integrating these models with techniques such as dynamic voltage frequency scaling (DVFS), resource virtualization, switching to low-power states, database partitioning, or even shutting down unused servers can further improve data center energy efficiency.

3.1.2 Query Processing and Query Optimization

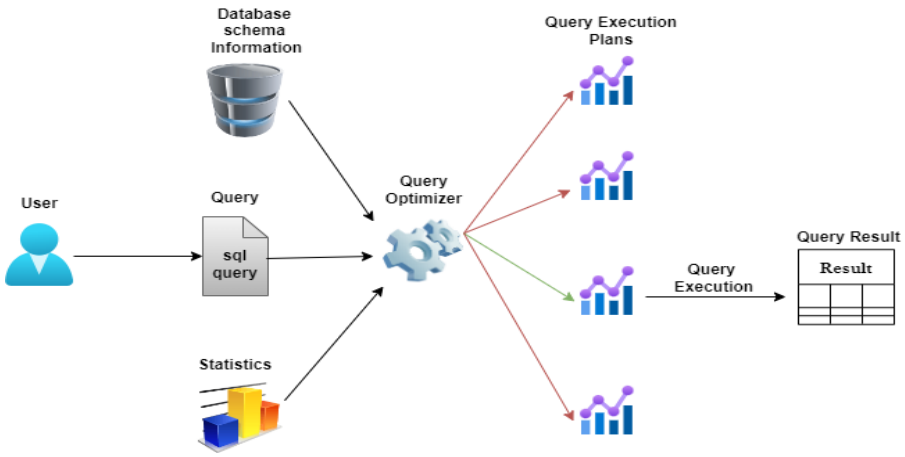


Figure 2: Query optimization-execution plan in DBMS

[2]https://www.inrae.fr/en/centres/clermont-auvergne-rhone-alpes
[3]https://tscf.clermont.hub.inrae.fr/

To analyze the research scope for optimizing the energy consumption of database systems, several modules are considered, including query processing, query optimization, data modeling, and the configuration of cache structures, cloud patterns, consistency levels, and latency. Most studies have focused on query processing and query optimization in relational databases, Fig. 2 describes stages in DBMS to execute a query.

Query optimization is the process of selecting the most efficient query execution plan from many possible execution plans that could produce the same result. This involves considering factors such as optimizing queries, index usage, join order, and access methods to minimize resource usage (like CPU, memory, and disk I/O) and improve query performance. The main goal is to develop strategies for optimization that not only enhance query performance but also reduce energy consumption.

On the other hand, query processing transforms a high-level query expressed in a query language (like SQL) into a series of low-level operations executable by a database system. Quantifying energy consumption during query execution encompasses tasks such as data retrieval from storage, processing, and results retrieval. This analysis helps to understand the energy usage of queries in a database. The primary objective is to identify energy-intensive operations during queries and explore optimization avenues to reduce overall energy consumption during execution.

3.1.3 Cost Models

Traditional query optimizers prioritize query execution plans based on time cost, aiming to minimize the total time spent on CPU, disks, and communication channels to retrieve output for end users. However, to achieve power-efficient query execution, a new approach is needed, involving the development of a power model to estimate the power consumption of different query execution plans. By integrating power consumption into the optimization process, the query optimizer can generate query plans that minimize both execution time and overall power consumption.

The process begins with building a power model for query execution plans, enabling estimation of power consumption for any plan considered. By exploring the query plan space and comparing power consumption metrics, the optimizer can identify and select the most power-efficient query plan. To streamline this search and avoid evaluating redundant queries, a defined set of possible query plans are generated and assessed for power efficiency. Fig. 3, shows the integration of the power cost model in the query execution process.

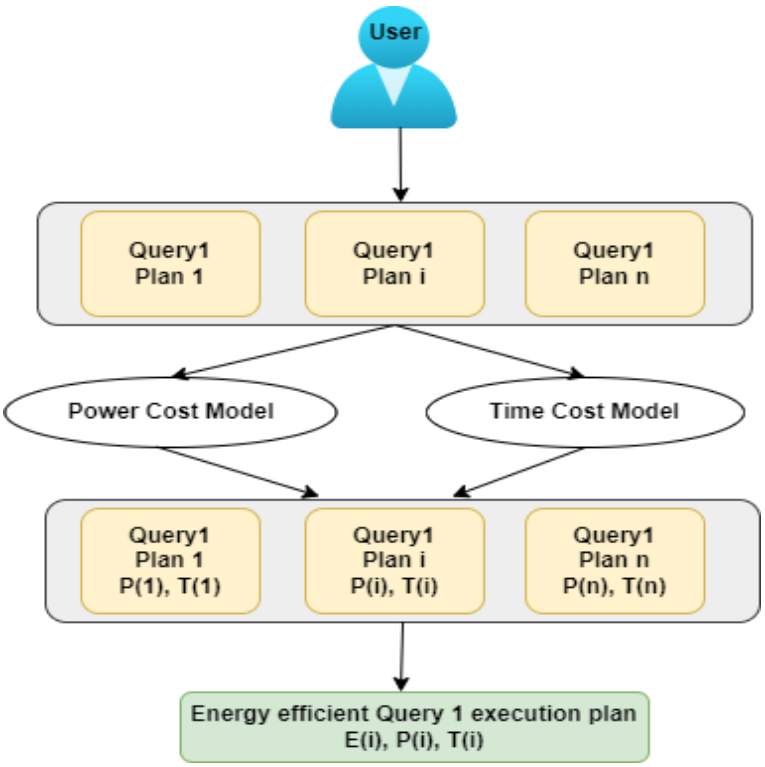


Figure 3: The process of integrating power cost model into query optimization

Most of the existing studies develop an analytical model that balances both time optimization and power efficiency. This model abstracts the power consumption of hardware components into system parameters, which are learned and integrated with a straightforward operator model. The operator model utilizes database statistics and query plans to estimate key parameters such as CPU instructions, memory accesses, and disk read/write requests. These parameters are essential for the hardware abstraction model to accurately predict energy costs during query execution.

Most of the papers described in the related work section use cost models developed with multiple-linear regression techniques(used to model the relationship between one dependent variable and two or more independent variables) to measure the power and time of query execution plans. These models help in finding parameter values that allow for the selection of the best execution plan and reduction of the database's energy consumption. Costs are normally treated as the product of the number of basic operations needed in the query plan to process a query and the resource consumption of each operation.

Power cost model:

$$Power(Q_i) = [\beta_{\text{cpu}} \sum_{j=1}^{m_i} \text{CPU\_COST}_j] + [\beta_{\text{io}} \sum_{j=1}^{m_i} \text{IO\_COST}_j]$$

Time cost model:

$$Time(Q_i) = [\alpha_{\text{cpu}} \sum_{j=1}^{m_i} \text{CPU\_COST}_j] + [\alpha_{\text{io}} \sum_{j=1}^{m_i} \text{IO\_COST}_j]$$

In [7], they describe Time(Qi) as the execution time of query i, Power(Qi) as power consumption of query i, IO_COST is the predicted number of I/O operations (pages) required for executing a specific operator, CPU_COST is the predicted number of CPU cycles to be processed in the CPU needed to run for a specific operator.

α_{cpu}, α_{io}, β_{cpu}, β_{io} are unit power cost parameters estimated while learning the model from training data. To estimate these parameters, a series of observations is performed using well-chosen queries. During the execution of these queries, the power values consumed by the system are collected using measurement equipment(Power meters, listed in Table 2). Simultaneously, the I/O and CPU costs for each training instance are calculated.

α_{cpu} is the estimated CPU time required to execute one CPU cycle needed to run a specific operator, α_{io} is the estimated time needed for the device to execute one I/O operation (process one page from disk to memory), β_{cpu} is the estimated CPU power required to execute one CPU cycle needed to run a specific operator, β_{io} is the estimated power needed for the device to execute one I/O operation (process one page from disk to memory)

3.1.4 Relational operators in DBMS optimizer

While executing a query on a data server, the query optimizer generates plans with an efficient sequence of operations to execute the query. Below we describe some of the operator types provided by PostgreSQL:

- **Sequential scan:** It scans each row of the data table sequentially, and analyzes relevant columns according to the predicate afterwards.
- **Index scan:** The relation is scanned by using an established index(tree-based or hash), which reduces the number of tuples accessed.
- **Bitmap scan:** A bitmap scan generates a bitmap that marks the positions of tuples meeting a specific query condition. For queries with multiple conditions, individual bitmaps are combined using bitwise AND or OR operations, resulting in a final bitmap that identifies tuples satisfying all conditions. These tuples are then retrieved from the database for further verification of the conditions.
- **Sort:** A sort operation arranges the rows of an existing result set in a specific order based on one or more columns.
- **Aggregation:** The aggregate operator combines multiple values to produce a single result for a group. Examples include calculating the average of a set of values or counting the number of tuples within the group.

- **Merge/Hash join:** Tables can be joined using either a merge join or a hash join algorithm. A merge join requires the tables to be sorted on the join attribute before joining. In contrast, a hash join does not require sorting but uses a hash table on one of the tables to quickly find matching tuples.
- **Nested loop join:** A nested loop join combines two tables by iterating over one table with an outer loop and over the other table with an inner loop, effectively checking the cross product of the tables to find matching tuples.

While developing cost models, we need to determine unit cost parameter values for all these operators. By analyzing performance with training workloads, we can develop accurate cost models. In the next section, we describe related studies on energy optimization that used similar cost models.

4 Related Work

The study aims to conduct a comparative analysis of existing optimization methodologies for energy consumption in database systems. Subsequently, we propose, implement, and evaluate an advanced optimization methodology to reduce energy consumption across large datasets. Our research into previous works reveals several scenarios for developing methodologies to reduce energy consumption in databases and enhance their energy efficiency. We describe a few below by classifying them into, section 4.1 Query Processing and Query Optimization on Relational databases, section 4.2 Query Processing and Query Optimization on Non-relational databases, section 4.3 Index and Materialized view selection on databases to enhance energy efficiency.

4.1 Energy consumption analysis for Query Processing and Query Optimization on Relational databases

In [9], the authors presented an approach to query optimization and processing, with a focus on energy efficiency. It begins by acknowledging that queries typically have a predefined response time goal often dictated by Service-level agreements (SLAs). Their proposed framework shifts the optimization problem towards finding and executing the most energy-efficient query plan that meets these SLAs. The operator model utilizes query plans and database statistics to estimate query parameters for basic operations like selection, projection, and joins. Additionally, an analytical model is developed to estimate the energy cost of executing a query. The developed energy consumption model alongside the conventional response time model, resulted in the generation of an Energy response time profile (ERP) for each query. By integrating detailed ERPs into the optimization process, the framework aims to select the most energy-efficient plans while meeting SLA constraints. The authors presented results for two join queries using modified Wisconsin Benchmark tables, with Query-A involving a 10% selectivity join on small 1GB tables and Query-B being a full join on sorted keys of larger 5GB tables. Experimental results highlight improvements in energy efficiency when selecting plans with lower power consumption, along with acceptable performance penalties. However, in their developed model they used only an important subset of database operations; it may not apply to certain types of queries and may not cover all query operations.

In [10], the authors focus on understanding and optimizing power utilization in database engines, with a particular emphasis on peak power consumption. They describe an approach to analyze peak power cost for both multiple queries and a single query utilizing a pipeline-based model. This model accounts for the power cost of each operator, determined by the highest rate of data flow from upstream operators. To predict peak power for query plans, they introduce a peak power estimator algorithm grounded in a regression model crafted from carefully chosen training examples. However, estimating peak power is complex due to the concurrent execution of pipelines, which are continuous sequences of operators. The parameters used in the regression model are categorized into Leaf Pipelines and Internal Pipelines, dissecting the interplay of operators, data rates, and the sizes of incoming and outgoing data. Queries from TPC-DS and TPC-H benchmarks are considered to generate diverse pipeline models. After removing power-insignificant operators, a set of "power-distinct pipelines" is obtained, capturing both intra-operator and inter-operator parallelism. However, to model the peak power behavior of a pipeline, an impractical number of training instances are going to be computed, which need a lot of pre-works to compute and which may need months in terms of time. Despite this challenge, various strategies are employed to generate them, including altering the size of scanned relations, selectivities of inputs,

and join conditions. But if the size of the data is large, it will inevitably take more time. Additionally, blocking operators cannot be pipelined, as they require their entire input before producing any output.

In [11], the authors proposed a methodology to enhance the power efficiency of query execution plans in databases by redesigning the query optimizer. The methodology involves several steps: first, building a power model to estimate the power consumption of query execution plans; then, introducing a possible query plan set and an algorithm to generate it, reducing redundancy; and finally, searching within this set for the most power-efficient query plan. They designed a power model to estimate the power consumption associated with executing query plans, trained using data collected from executing query plans with different basic operations and dataset sizes. An algorithm is proposed to generate power-efficient query execution plans based on this model. Experimentation using the TPC-H benchmark evaluated the accuracy of the power model and compared the power-efficient query execution plan with traditional optimal plans. However, the paper lacks adaptable power models suitable for dynamic environments in real-world database systems. Additionally, scalability and generalization are overlooked, with experiments conducted on small database sizes (500MB, 1GB, and 5GB), highlighting the need for evaluation across varying database sizes, query complexities, and concurrency levels to ensure practical relevance. As per the graphs, Execution time has increased for power-efficient execution plans.

In [12], the authors proposed a method to predict the energy cost of query plans before execution based on their resource consumption patterns during query processing. Additionally, a query plan evaluation model is introduced to assist the query optimizer in selecting plans that meet performance requirements while resulting in lower energy costs. Using the cost model as a foundation, the evaluation model leverages the trade-offs between the power and performance of plans. To construct an accurate energy model, they conduct an extensive study of the effect of memory size and cache structures (cached data) on the performance and power consumption of query processing. Three main cache structures (Database Buffer cache, Dictionary cache, and Library cache) in memory, associated with IO and CPU resource consumption, are considered in their study. They compare power consumption values with different cache pairs, and based on these experimental values, they develop cost models to measure the energy cost of database operations. They used the TPC-C benchmark as a training set to obtain values of key parameters of the cost model and the TPC-H benchmark as a test set to evaluate the cost model. With a workload having 'n' number of plans to execute queries, they reduce plans to 'n-m' by setting a threshold value for query performance, then calculate the energy cost for these plans using the energy cost model, and finally select one plan. However, they manually rewrote query statements with hints rather than relying on the original queries. This approach may introduce complexities and potential risks, such as the need for ongoing maintenance and the possibility of unintended consequences if hints are not properly implemented. Performance degradation % increases from 3.3% to 12% as the database size is increased from 1GB to 10GB.

In [13], the authors proposed a methodology to estimate the power and energy costs of database servers. Rather than measuring the power and energy consumption of individual hardware components, they focused on measuring the total power and energy consumption of a server. The methodology involved using multiple-linear regression, and a set of selection queries based on TPC-H as training query workload to derive cost models that are based on readily available workload statistics such as selectivity factors, tuple size, number of columns, and relation cardinality. They developed cost models to measure the power and energy consumption of servers. The experiments were conducted on four server machines to examine whether energy consumption is proportional to the number of servers. They found that the relation between energy and the number of servers is indeed proportional. Additionally, they observed that in the case of peak power, this relationship was inversely proportional to the square of the number of servers. Peak power was influenced by selectivity, the number of columns in tables, and the number of servers used in a multi-server configuration. The authors validated their experimental results by comparing them to alternative methods[14]. The findings indicate that their approach yields more accurate predictions for power and energy.

In [14], the authors argue that traditional query optimizers primarily focus on minimizing total processing time by prioritizing I/O operations. However, optimizing solely for I/O may not lead to optimal power efficiency. Instead, the authors propose considering both power and performance in query optimization in DBMS's to achieve better results. Integrated a power cost model to the modified query optimizer can evaluate and compare the priority among alternative plans of a query, and finally select an optimal plan towards a user-preferred optimization goal. To calculate the power cost of each plan, the study utilizes a power profile of basic operations combined with an operation vector, mapping each plan to the number of basic operations needed for computation. This allows for the estimation of power

consumption for each query plan. The model introduced allows for adjustable trade-offs between power consumption and query execution time, with a coefficient 'n' representing the relative weight placed on these factors. The study suggests that exploring a larger search space for each query can identify more opportunities for such trade-offs, leading to substantial power reduction. However, it also notes the limited power-saving potential in the current method for single-table query scheduling, and the study includes the use of a static energy model, which may not fully capture the dynamics of system behavior.

To enhance the work in [14], authors continued their experimentation and modified models as discussed in [15], [16]. They provide a broader discussion on the energy profiles of key relational operations and various database maintenance operations. Additionally, they investigate how their energy models impact query optimization through an energy efficiency study. Due to page restrictions, we do not provide detailed descriptions of these studies here.

Table 1 below provides an overview of the related works described in this section, highlighting the percentage of energy they were able to optimize using their models in each study, along with the rate of accuracy of their developed models to measure energy consumption and some limitations from the papers.

Accuracy: It defines how accurate the estimated power is compared with the actual measured power. In some papers, they validated the accuracy of their developed models by comparing calculated and measured values of energy, and in others, they validated by comparing their cost models with cost models in other papers.

Paper	Energy Opt.	Model Accuracy	Remarks
[9]	>10%	3 to 8%	SLA, energy model developed by analysis of only a few operators (selection, projection, join)
[10]	N/A (peak power cal.)	3 to 12%	Significant time to analyze which operator is consuming more power in different query plans
[11]	less than 5%	5.2 to 12.1%	developed models by analyzing parameters on very less database size(500MB), Perf degraded as database size increases
[12]	7.7 to 18%	5.63 to 5.92% improved compared to [16]	3.3 to 12% of time perf degradation increases as database size is increasing
[16]	14.7 to 33.3%	10 to 13.7%	8.3 to 13.5% execution time degradation as database size is increasing
[14]	upto 19% for trade-off query plans	7.2 to 14.5% for indv.operators (Seq and Index scan)	exe time increases from 48 to 243 minutes as database increases from 1GB to 10GB

Table 1: Summary of related works on relational databases

4.2 Non-relational (NoSQL) databases

4.2.1 Comparison between Relational and Non-relational databases

From papers [17], [18], [19] the choice between SQL and NoSQL databases depends on various factors; including the specific requirements of the application, data consistency needs, scalability requirements, and the expertise of the development team. But, as data volume increases; relational databases can face scalability challenges and may not be optimized for handling unstructured or semi-structured data efficiently. NoSQL databases offer high data efficiency, horizontal scalability, and flexible schema design, making them well-suited for distributed systems designed to work in clusters. NoSQL databases can provide good performance for write-heavy workloads and can be more cost-effective in terms of storage space due to their ability to scale horizontally, and a query doesn't have to look at several tables to get a response. Depending on data requirements; we can store data in either form of Document data stores (MongoDB), column-oriented database (Cassandra), Key-value data stores (Redis), Graph data stores (Graph QL), etc. By considering these factors, we reviewed several existing papers that compare both SQL and NoSQL databases in terms of query performance, and power consumption, and also provide comparisons between different NoSQL databases to gain an overall understanding of database performance.

In [20], authors evaluated the performance of PostgreSQL, MongoDB, and Cassandra using a social media web application. The first test assessed performance with 100 users over 5-minute intervals, involving actions like logging in, viewing posts, and creating new posts with hashtags. Databases were tested with four datasets: 1000 users with 1 million posts, 5000 users with 5 million posts, 10,000 users with 10 million posts, and 15,000 users with 15 million posts. PostgreSQL was slowest in reading posts from a timeline, while MongoDB was most efficient for large datasets. The second test involved inserting 1000 records, where MongoDB was the slowest and PostgreSQL was the most effective. The final test evaluated searching hashtags, with Cassandra being the slowest and MongoDB approximately twice as fast as PostgreSQL. Overall, MongoDB excelled in reading and searching data, while SQL databases were the fastest for writing data.

In [21], authors evaluated the performance of both relational databases (Oracle, MySQL, SQL Server) and non-relational databases (MongoDB, Redis, Neo4j, Cassandra). They compared SQL and NoSQL databases based on factors like data consistency, scalability, and workload suitability. The advantages of NoSQL databases in terms of flexibility and scalability for handling unstructured or semi-structured data and write-heavy workloads were highlighted. Differences between SQL databases were described in terms of supported programming languages, transaction control, and replication strategies. Experimental results compared the performance of different databases on query operations, focusing on speed and efficiency. They tested a database of train connections in Slovakia with datasets of 10,000 and 90,000 records using indexing and range scan operations. Experiments conducted on a MacBook Pro (2015, 8GB RAM, Intel Core i5) showed that non-relational databases, especially MongoDB, outperformed relational databases due to faster read/write operations to a single entity.

4.2.2 Energy consumption analysis for Query Processing and Query Optimization on Non-relational (NoSQL) databases

In [22], authors monitored the energy consumption of queries using aggregate and map-reduce operations on MongoDB. They analyzed energy consumption for Insert operation by using TPC-H queries with and without index, map-reduce, and aggregate pipeline functions. Their results showed that the aggregate pipeline is more effective than map-reduce, and indexing improves query performance. In MongoDB, query processing involves planning and execution phases. During planning, the system determines the most efficient way to execute a query. During execution, the system follows the query plan, using indexes to accelerate retrieval. Power consumption of queries was measured using the RAPL Power Meter, integrated with jRAPL. Results and power consumption data were returned in a JSON file. They suggest further testing with different document store systems and varied application contexts to validate their findings.

In [23], authors demonstrated the energy consumption of queries in data centers using a dashboard populated with data from a Microsoft-provided sample database. They assessed one relational database (MySQL) and one non-relational database (Neo4j). They designed 14 distinct queries and converted SQL queries to Cypher for Neo4j. They extended the gSQL [24] (which categorizes SQL queries based on their energy efficiency) tool to handle both SQL and NoSQL queries, and measured energy consumption by using the intel jRAPL framework. To obtain energy data, they defined specific parameters: a configuration file with DBMS connection details, an input file with queries, and the number of repetitions for each test. Each query was executed 15 times, and the average energy consumption and execution time were analyzed. They concluded that MySQL is more efficient than Neo4j in terms of performance and power consumption. However, the conclusions are not fully aligned due to factors like the suitability of the dataset and queries for relational databases, and the complexity of converting queries from MySQL to Cypher. Additionally, the authors did not discuss the specific energy consumption of NoSQL components or functions.

In [25], they present a performance and energy consumption evaluation of NoSQL DBMSs, specifically Cassandra, MongoDB, and Redis. Experiments are based on the YCSB benchmark, and the results demonstrate that energy consumption can vary significantly for different workloads. Various workloads generated by the Yahoo! Cloud Serving Benchmark (YCSB), and insert, read, and delete commands are evaluated for each DBMS. A measurement framework, namely EMeter, is introduced to facilitate the collection of data related to energy consumption and execution time in the DBMS server. The framework encompasses hardware components and a software tool to store and visualize the collected metrics. For management and control, EMeter includes a Graphical User Interface (GUI) through which real-time consumption and execution information can be visualized. Overall, no DBMS exhibits dominant behavior in terms of both execution time and energy consumption across all workloads. they used default

configurations for all experiments, and tuning could yield different results. By comparison, MongoDB maintains stable performance across all workloads, although it doesn't dominate in any specific scenario for the adopted system.

Since NoSQL databases are designed for large-scale data storage and high-concurrent data processing, their scalability also implies significant energy consumption, thus demanding higher energy efficiency and proportionality. In this research, the authors identify Waiting Energy Consumption (WEC) as a factor contributing to energy wastage due to computer idleness in NoSQL databases. WEC occurs when certain nodes in a cluster remain in a "passive idle" or "busy idle" state while waiting for other resources. They propose a novel approach to reducing WEC to minimize energy wastage in NoSQL databases. The study [4] analyzes WEC across four NoSQL databases (HBase, Cassandra, HadoopDB, and Hive) and five types of queries (loading, fuzzy search, range search, aggregate, and join). It reveals that in distributed cluster systems, where jobs are scheduled across different nodes, some nodes may experience delays due to scheduling issues or insufficient incoming jobs. Additionally, NoSQL databases employing a non-relational data model often involve high local and network I/O operations, making them I/O intensive. This results in CPUs waiting for extended periods, leading to increased waiting energy consumption. Many NoSQL databases utilize the Map-Reduce model for query operations, including selection, aggregation, and fuzzy selection. However, improper implementation of Map-Reduce can lead to poor parallelism and synchronization, exacerbating waiting energy consumption. One suggested solution to reduce WEC is to shut off idle systems, but this may not be feasible in distributed NoSQL systems where nodes cannot be shut off while waiting for job scheduling, I/O operations, or computational results from other nodes. Instead, the authors propose reducing waiting energy consumption by minimizing network I/O, synchronizing CPU and I/O operations, and selecting appropriate Map-Reduce frameworks based on data characteristics. They conduct experiments using their data and benchmarks. Although they introduced Waiting Energy Consumption, a method for monitoring waiting energy remains unknown.

In [8], the authors conducted a comprehensive study on the impact of query optimizations on performance and energy efficiency in relational (MySQL) and NoSQL (MongoDB, Cassandra) databases using a customized dataset from 100GB of twitter data and the YCSB benchmark. Experiments were run on the NSF-funded high-performance computing system 'Marcher', utilizing an API to collect real-time power data. For MongoDB, they observed significant performance and power differences in optimized queries(Covered Query vs. un-optimized query, Non-indexed vs. Indexed query, Ordered vs. Unordered query, Projection optimization using Aggregation, and Sharding), except for ordered query optimization which showed minimal power difference. Sharding improved performance threefold but doubled power consumption. For Cassandra, three query optimizations were tested (Row caching, STCS, LCS), with minimal power differences and LCS providing the best performance improvement. In MySQL, indexing accelerated execution and reduced power consumption, achieving over 20% energy savings by specifying column names instead of using "select*", and the EXISTS operator also improved query performance compared to IN.

Cross-database comparisons using the YCSB benchmark showed NoSQL databases outperforming MySQL, with MongoDB as the best performer. Using Twitter data for common queries, MongoDB consistently performed better than the other databases. Also, the impact of Dynamic voltage and frequency scaling (DVFS) was analyzed by using three technologies Performance, Powersave, and Ondemand, with the "ondemand" governor state providing the best balance of performance and power consumption across all databases. However, tests were limited to single queries executed only once, and details on implementing the power measurement tool on the Marcher system were lacking, limiting the reliability of the conclusions.

Table 2 shows a list of devices used in various studies to measure power consumption in databases for individual operators during the development of cost models.

Paper	Monitoring Device
[9]	Yokogawa WT210 unit
[13]	power meter (didn't mention specified name)
[10]	digital power meter (Brand Electronics model 20-1850/CI)
[11]	PM 1000+
[12]	WT3000 digital power analyzer (YOKOGAWA)
[14], [15], [16], [26], [27]	Watts up? PRO ES power meter
[22], [23]	RAPL/jRAPL framework
[25]	EVM430-F6736 hardware / Emeter software
[4]	Power-meter
[8]	API/Marcher system/Log power to file API

Table 2: Power monitoring devices used in existing methodologies

Table 3 provides an overview of related works described in this section focusing on non-relational databases, highlighting the query operations they examined in each paper, platforms (DBMS) used for performance and energy consumption analysis, and the conclusion they provided by comparing SQL and NoSQL databases.

Paper	Level	Platform (DBMS)	Conclusion
[22]	Single Query by using Map-Reduce and aggregate functions	MongoDB	Use of the Aggregate pipeline functions is more effective than Map-Reduce
[23]	14 read-only queries	Neo4j, MySQL	Indeed SQL performs well because of SQL friendly queries
[25]	Insert, Delete, Update workloads	Cassandra, MongoDB, Redis	MongoDB maintains stable performance across all workloads compare to remaining databases
[8]	Single Query (Insert, Update, Search)	MongoDB, Cassandra, MySQL	MongoDB performs well compared to remaining databases among overall operations
[20]	Search, Upload, View timeline Posts	PostgreSQL, MongoDB, and Cassandra	MongoDB excelled in reading and searching data, SQL database fast for writing data
[21]	Select, Insert, Delete, Update	SQL and MongoDB	MongoDB well performed compared to relational databases due to faster read/write operations to a single entity

Table 3: Summary of related works on Non-relational databases

4.3 Index and Materialized view selection on databases

Increasing the energy efficiency of database systems is possible by focusing on the physical design of the systems. This involves developing optimization structures such as creating indexes and materialized views for a given workload. These optimizations aim to enhance query performance and minimize energy consumption within database operations. Few existing research methodologies address the creation of these optimization structures.

In [28], authors propose a tool for automatically creating indexes for a given workload. The paper aims to address the challenge of selecting an optimal set of indexes for a given database and workload, considering various index types such as single-column or multi-column, and clustered or non-clustered. The authors introduce a comprehensive index selection tool architecture comprising a basic search algorithm and three key modules: candidate index selection, configuration enumeration, and multi-column index generation. Efficiency metrics for the tool include the number of indexes and configurations considered, as well as the number of optimizer invocations needed. Techniques are proposed to reduce optimizer calls by introducing the concept of atomic configurations, where a query's execution can utilize all indexes within the configuration. The paper outlines strategies for identifying atomic configurations, starting from one index based on query processor characteristics and another adaptive approach based on index interaction. Additionally, optimizations are introduced to accurately estimate the cost of queries for non-atomic

configurations by extending costs from atomic ones. Experimental validation using the TPC-D workload includes metrics such as the number of candidate indexes selected, optimizer calls, atomic configurations, and index creation time, but lacks assessment of index utility during workload execution.

In [29], the authors introduce a tool AISIO, an Automatic index selection integrated into optimizer(AISIO) as a novel approach for improving database performance without external tools. AISIO seamlessly integrates with the PostgreSQL, employing a heuristic approach to guide index selection. Its architecture includes modules for workload submission, statistics collection, heuristic analysis, hypothetical index creation, optimization, and recommendation. AISIO's heuristic involves a four-step process to identify candidate indexes based on SQL statement characteristics, operating internally within the DBMS to recommend potentially beneficial indexes for transaction performance improvement. Integrated tightly with PostgreSQL, AISIO accesses the DBMS data, enhancing its effectiveness. However, reliance on the DBMS optimizer for index evaluation may introduce overhead, particularly in high-throughput or concurrent transaction environments. The dependence on the DBA's decision-making for index creation can also cause delays and suboptimal index utilization, and AISIO's reliance on the DBMS's cost model may not fully capture all aspects of query performance, potentially affecting the accuracy of index recommendations.

In [7], the authors present an initiative for eco-physical design in the database lifecycle, aiming to optimize both query performance and energy consumption during the physical design phase. Their approach considers a workload (W), constraints (C), and non-functional requirements (NFR) to produce Materialized views (MV). A materialized view is a database object that contains the results of a query and can be refreshed periodically. Unlike a regular view, which is a virtual table generated dynamically when the query is executed, MV stores the query results physically for faster performance. The selection process of MV involves constructing a global query graph (GQG) with nodes tagged for energy cost and query benefit and is NP-complete due to the exponential growth of the solution space. The cost models balance performance and power consumption, using evolutionary algorithms (EAs) and Pareto ranking for optimization.

The proposed method has several limitations, changing the server or database configuration affects the accuracy of evolutionary algorithms for certain queries. The trade-off materialized views (MV) balancing time and power as trade-off values. These trade-off solutions typically require 70% to 100% of the available storage space, with optimal solutions emerging when storage constraints are relaxed. The impact of maintenance costs is significant; while a 50% allocation for maintenance results in efficient trade-off solutions, the best results are achieved when this allocation is further relaxed. Ongoing experiments aim to assess the scalability of the solution, focusing on performance under very large workloads and datasets.

5 Proposed Solution

While examining the existing approaches to energy-efficient database design, it becomes evident that numerous strategies exist for query optimization and processing. Each of these implementations shows off its unique set of advantages and limitations, determined by the specific workloads they handle, and the available resources at their disposal. Across all studies, the operator's performance metrics are typically measured using training data, this helps in creating cost models that include specific parameters relevant to each process.

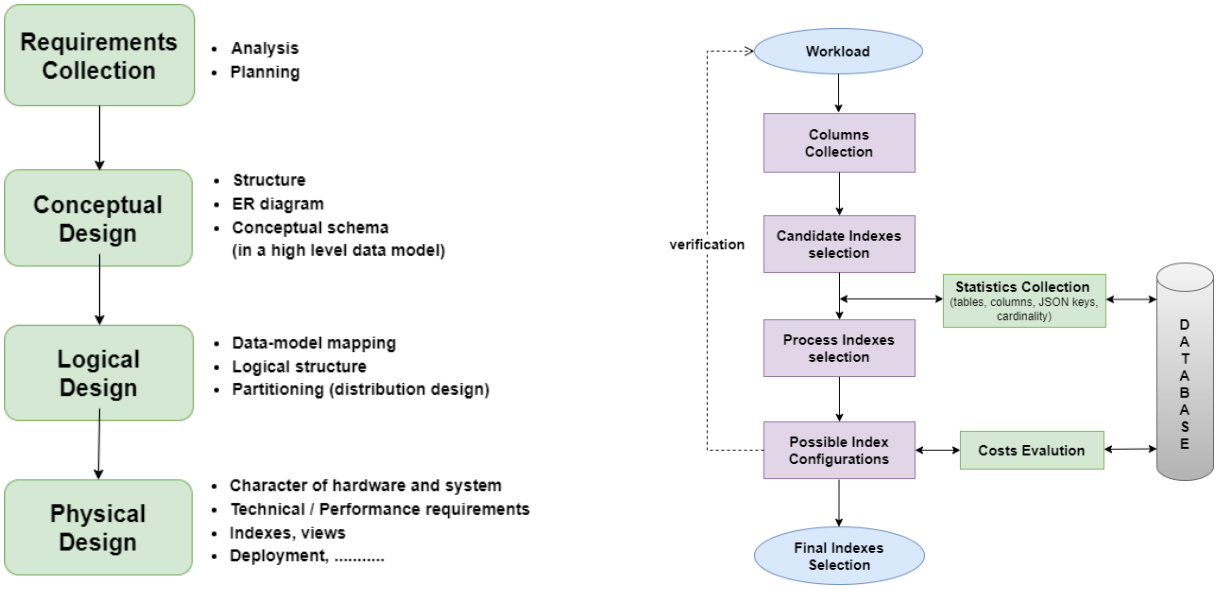
In [10], authors mentioned that using an index scan consumes lower peak power with acceptable response-time penalties compared to hash-join operators. In [8], it is shown that using indexes on SELECT and DELETE operations significantly reduces time and power consumption, while for INSERT operations, the execution time remains the same but power consumption decreases. However, these findings are based on executing a single query only one time on both relational and non-relational databases. In [30], authors describe various types of indexes on databases and demonstrate how they efficiently improve only query performance, and they do not consider the power consumption of the databases. In [28], [29], detail the process of selecting indexes for a workload and present metrics such as the number of candidate indexes selected, optimizer calls, atomic configurations, and index creation time, but they do not evaluate the utility of indexes on the workload.

Also, while most of the data centers use indexes to enhance query performance, there is a lack of analysis on how effective these indexes are in optimizing energy consumption. It is widely known that indexes primarily benefit read operations, often at the expense of write operations, which can

experience degraded performance. Therefore, it is crucial to measure the extent to know how indexes contribute to energy optimization. Considering all these factors from previous papers, we chose to develop a methodology to select a set of indexes for a given workload, and to measure how much energy consumption and performance is optimized by using these indexes during the physical design phase (shown in Fig. 4a) of database management.

In the context of databases, a workload refers to the set of queries and transactions that are executed by the database system over some time. This workload can include various types of SQL operations such as select, insert, update, and delete statements. Workloads are important for understanding the performance characteristics of the database and for making decisions about optimization structures, such as index selection and creation.

Fig. 4b illustrates the architecture of our selected solution. We experimented with our methodology using different workloads to evaluate how efficiently these indexes benefit the database. We developed an algorithm by giving training query workloads to it to check its efficiency at each stage. We measure the performance and energy consumption of the workload on a database with and without using indexes to determine how effectively these indexes optimize databases for different workloads (read, read-write, write). Furthermore, a user interface is necessary in our process to facilitate interactions and allow users to specify their desired optimization structures (indexes) for a given workload. This approach ensures that energy efficiency is considered alongside performance in the management of database systems.



(a) Database design life-cycle

(b) Architecture of selected solution

Figure 4: Database life-cycle levels and Architecture of proposed solution

1. Workload analysis: The process of selecting index columns begins by analyzing the workload ready to run on the database. We examine the queries within the workload, extracting all columns and JSON fields to provide a starting point for the index selection process.

Next, we retrieve tuples that show how many times each column and JSON field is used in the entire workload and how many queries use these columns, represented as [Column, Count in workload, Number of queries using the column]. We select candidate index columns from index columns by applying a condition that selects columns that are repeated more than 'n' times in the workload.

2. Cardinality statistics: We retrieve these candidate index columns along with their cardinality statistical data obtained from the database. This helps us understand their presence, sizes, and distributions in the database, formatted as [Column, Count in workload, Number of queries using the column, Cardinality of column].

3. Identifying process index columns: We identify process index columns from the candidate indexes by applying a condition that selects columns with a cardinality count of more than 'm' in the workload. Subsequently, we explore all potential indexes from these process indexes, usually focusing on combinations of up to two columns.

4. Evaluating index configurations: To evaluate the final indexes from all possible indexes, we proceed with one of the following methods:

4(a). Query history analysis: We analyze the database query history, retrieved by using pg_stat_statements to identify frequently used queries and columns. Based on this analysis, we select the most frequently used columns for index creation.

4(b). Test environment deployment: We deploy a test environment, often a replica of the original database. Here, we assess the execution time and energy consumption of each index by executing the workload on test environment data. The results of this analysis guide our selection of indexes.

5. Final indexes selection: Currently, we use method 4(b) to evaluate indexes based on analyzing time and energy consumption in the test environment, and from the insights gained from 4(b), we select indexes that promise better execution time and energy optimization for the workload. We evaluate the final selected indexes by executing the workload on the database both with and without the indexes.

To begin, we execute a workload consisting of read-only queries on the database. Using our methodology, we identify the final indexes (or index) and proceed to create these indexes (or index). Subsequently, we execute the workload on the indexed database and record the time and power consumption, allowing for comparative analysis against the workload executed without the index.

5.1 Development Stack and Tools

5.1.1 CEBA Platform

We implement our methodology on the Environmental Cloud for the Benefit of Auvergne (CEBA) platform[4]. It is a specialized cloud-based infrastructure designed to support agricultural activities by providing environmental data and analytical tools. Its primary goal is to enhance agricultural sustainability by leveraging environmental data such as weather conditions (temperature, humidity, precipitation), soil moisture, geospatial data, and other relevant parameters.

The platform employs a combination of cloud storage, database systems, and data lakes to store diverse data types, ensuring scalability and accessibility for various users and applications. CEBA offers a rich dataset for researchers to develop new agricultural technologies and practices. It stores data received from a network of sensors, making it specifically designed to handle time-series data, which is typically generated by sensors.

5.1.2 PostgreSQL

PostgreSQL[5] is an open-source database management system (DBMS) that allows the storage of a large number of different data types and has a very active community. It can manage JSON data. Within CEBA, the sensor database is organized around its associated sensor networks, employing various schemas tailored to the received data structure. Presently, there are two schemas implemented: one for delimited files and another for the connecsens network utilizing JSON files. JSON has been adopted as the preferred format for managing IoT data on their platform, leading to its extensive usage within their database. Occasionally, there is a need to convert raw data files into JSON format. In PostgreSQL, both JSON and JSONB (JavaScript Object Notation Binary) are available for storing raw JSON data. JSONB was chosen due to its broader array of query and comparison capabilities compared to JSON.

5.1.3 Python

Python[6] is the primary programming language in our work due to its diverse libraries and frameworks that facilitate integration with PostgreSQL, store results in CSV files, draw plot graphs for the results in CSV files, and easily run workloads on PostgreSQL.

5.1.4 Environmental setup

Our experimentation environment runs on a robust setup featuring a 13th Gen Intel(R) Core(TM) i7-13700H processor clocked at 2.40 GHz and 16.0 GB of RAM. Operating on Windows 11 Professional 64-bit, our primary DBMS for experimentation is PostgreSQL 16. The database size stands at approximately

[4]https://www.mdpi.com/1424-8220/22/7/2733
[5]https://www.postgresql.org/
[6]https://www.python.org/

2.5 GB and 4.5 GB. This setup allows us to conduct all experiments on a single database server, ensuring efficient processing and management of our tasks.

6 Implementaton

Our implementation began with the development of an algorithm to extract columns from the workload to select indexes, following a step-by-step architectural approach. We developed this algorithm by using training workloads to ensure it accurately extracts all columns from different types of queries without missing any.

After identifying efficient indexes from the possible index configurations, we first ran the workload on the database without creating an index. We measured both the execution time and energy consumption of the workload. Then, we created the selected indexes in the database and measured the execution time and energy consumption again. Each workload was run five times before creating the indexes and five times after creating the indexes. We compared the average of the set of results to analyze the efficiency of the indexes in reducing energy consumption on the database.

Before measuring the time and energy consumption of workloads, we extracted costs and row counts from the query plan by running 'EXPLAIN ANALYZE' on the database. PostgreSQL offers a powerful tool for analyzing query plan cost factors, known as 'EXPLAIN ANALYZE[7].' By running a query with 'EXPLAIN ANALYZE' at the start, PostgreSQL executes the query and provides detailed information about how it was executed. This includes the operations used to execute the query, the costs of each part of the query, the number of rows processed, and the execution time of the query.

For example, consider the following query run against our database to get the number of frames in a particular JSON file for a given application,

```
SELECT COUNT(data)  AS FrameCountofFile
    FROM connecsens.json_montoldre_row
    JOIN connecsens.json_file
    ON json_montoldre_row.file_id = json_file.id
    WHERE json_file.file_name = 'data20240407.json';
```

This query accesses two tables in the schema and returns the count of data frames of a particular JSON file. Query plan in PostgreSQL is defined as a tree where the root node produces the final result. The child nodes have an operator type and further attributes required to match the query description dependencies between the nodes are represented by the tree structure. Fig. 5 shows the tree for a valid plan of the query described above. Note that both scans can be executed in parallel by using an automatically created index on the primary key of the respective tables.

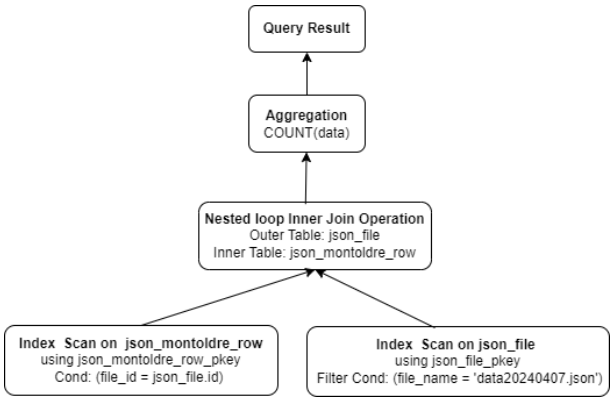


Figure 5: Query execution plan tree produced by PostgreSQL for a query involving a join on two tables and an aggregate function

The EXPLAIN ANALYZE statement produces a textual representation of how the query was executed, including the operations it used by displaying the nodes of the query execution plan tree as well as additional information explained below.

[7]https://www.postgresql.org/docs/current/using-explain.html

```
Aggregate  (cost=213313.45..213313.46 rows=1 width=8)
(actual time=2099.542..2099.544 rows=1 loops=1)
  ->  Nested Loop  (cost=0.71..213307.47 rows=2393 width=1108)
  (actual time=1754.791..2099.370 rows=528 loops=1)
        ->  Index Scan using json_file_pkey on json_file
        (cost=0.28..25.50 rows=8 width=4) (actual time=0.295..4.967 rows=8 loops=1)
              Index Cond: (file_name = 'data20240407.json'::text)
        ->  Index Scan using json_montoldre_row_pkey on json_montoldre_row
        (cost=0.43..26643.83 rows=1642 width=1112)
        (actual time=252.971..261.772 rows=66 loops=8)
              Index Cond: (file_id = json_file.id)
```

Costs: This is a measure used by the planner to estimate the resources needed for each step (including CPU, disk, I/O). It helps in deciding the most efficient execution plan.

Initial cost (Startup cost): The initial cost, also known as the startup cost, represents the estimated cost to begin executing a particular step in the query plan. This includes the cost of activities such as setting up data structures, loading indexes into memory, or initiating a scan.

Final cost (Total cost): The final cost, also known as the total cost, is the estimated cost to execute a particular step to completion. This includes the cost of processing all the rows expected to be retrieved by that step.

Rows: This is an estimate of the number of rows processed or produced by each step. It's based on table statistics and helps in understanding the scale of data involved at each step.

Planned rows: Planned rows refer to the estimated number of rows that the planner expects to be processed or produced by a specific step in the query plan. These estimates are based on table statistics and can influence the choice of the query execution path.

Actual rows: Actual rows indicate the real number of rows processed or produced by a specific step during the actual execution of the query. This provides important performance data for verifying the accuracy of the query planner's estimates and for optimizing query performance.

In PostgreSQL, these values from the query plan play a crucial role in understanding how indexes optimize time and power. They also help identify which queries in the workload are most affected by the indexes. So, before measuring actual values of the execution time and energy consumption of workloads we extract rows and cost values from workloads and we compare these results without actual time and energy values to see whether our actual results align with query plan results or not.

Before executing the workload on the database each time, we clear the cache data to measure real-time values of time and energy. We verified whether the cache is cleared or not, every time by using the 'pg_buffercache' extension in PostgreSQL, which is used to examine the shared buffer cache.

6.1 Results and Analysis

We used three distinct read-only workloads to evaluate our methodology, Fig. 6 represents its corresponding results for each workload from the index selection algorithm.

Column	Frequency	Query_Count	Cardinality
Workload1			
data ->> data-CNSSRFDataTypeName	19	13	18
data ->> data-node-timestampUTC	9	5	119165
Workload2			
data ->> data-temperature	33	20	732
Workload3			
data ->> servertimestampUTC	20	16	122098

Figure 6: Final indexes received for each workload from our index selection algorithm to analyze energy and time for each workload before and after using these indexes

Fig. 6 details the frequency of index in the workload (Frequency), the count of queries using the index (Query_Count), and the cardinality of indexed columns (Cardinality) for each workload. Our analysis involves evaluating the performance impact of using indexes compared to not using indexes for each workload.

6.1.1 Workload 1:

It consists of 20 distinct SQL queries that typically run on the database to retrieve results from specific applications, and environmental data types, and to access values received by various sensors. We can typically categorize the queries involved in workload1 are Aggregation queries, which focus on counting records or aggregating data based on specific criteria such as time intervals or device identifiers; Distinct value queries, which involve retrieving distinct values or pairs from the dataset, useful for identifying unique configurations or data types; Filtering queries, utilize conditions to filter and retrieve data based on specific attributes or criteria; Join queries, combine data from multiple tables to provide comprehensive insights into records associated with particular files or applications; Statistical queries, perform calculations or retrieve specific statistical information, such as counting records within the date ranges. Fig. 7 shows the total cost and row values of queries extracted from PostgreSQL query plan before and after creating indexes on the workload1.

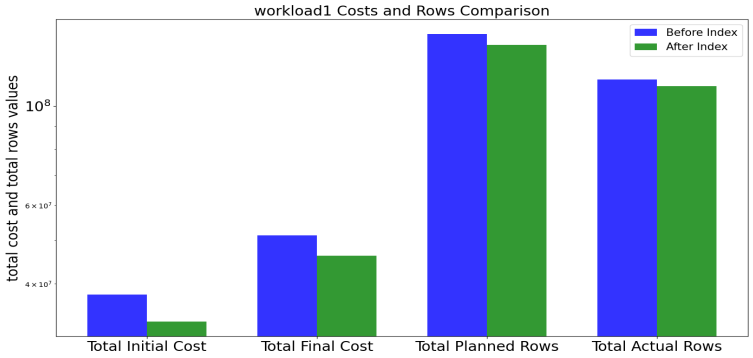


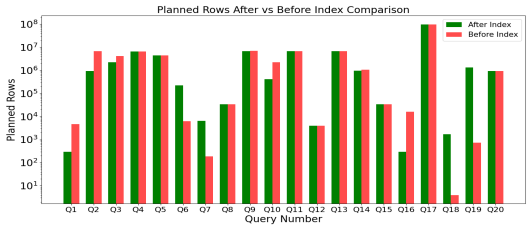
Figure 7: Workload 1 costs and rows comparison before and after creating indexes

By analyzing the individual query results, we found that the planned rows (shown in Fig. 8a) for specific queries [Q6, Q7, Q18, Q19] increased after creating indexes. In all these particular queries, the common point is the use of data `->>` 'data-CNSSRFDataTypeName' with a like '%Name%' condition in the WHERE clause. Before adding the index, the PostgreSQL query optimizer chose a sequential scan because no indexes were present. This led to a lower estimation of the number of rows that would qualify for the query conditions.

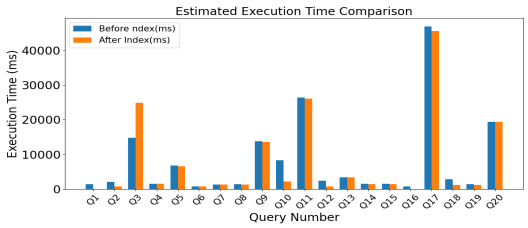
After adding an index on data `->>` 'data-CNSSRFDataTypeName' (assuming it's a partial index on 'TempDegC'), PostgreSQL adjusts its estimation based on the statistics and selectivity of the index. If the index is highly selective (meaning it filters out a large portion of the table), PostgreSQL might estimate a larger number of qualifying rows compared to before. If the index is not highly selective—perhaps because the filter conditions match a large portion of the table—the estimated number of qualifying rows might still be high. This can happen if the cardinality (number of distinct values) covered by the index is significantly lower compared to the total number of rows in the database. These adjustments can result in an increase in the planned number of rows before executing a query. However, these estimates are made by the query optimizer and do not directly correlate with actual energy consumption during query execution. The actual energy consumption of the workload is determined by the number of actual rows processed during query execution with and without index, which may not correspond to the estimated number of planned rows.

Additionally, the execution time (shown in Fig. 8b) sharply increases for Query 3, where we use COUNT and DISTINCT operators on the data `->>` 'data-CNSSRFDataTypeName' column. The performance of DISTINCT operator depends on the data distribution of the index column in the database. Since the DISTINCT operation needs to identify unique values, it requires sorting or hashing, which can be resource-intensive if the indexed column has a high number of duplicate values in the database. Similarly, the COUNT operation, while generally efficient with indexes, can become slow if it involves

scanning a large number of rows to count distinct values. In this specific case, the lower cardinality of the indexed column means that the index is not as selective, resulting in more rows being processed and longer execution times for Query 3.



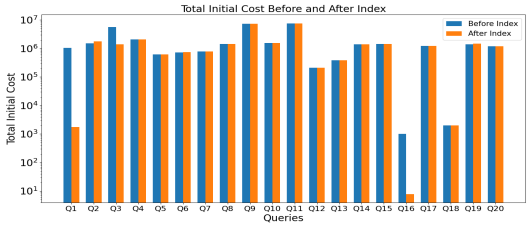
(a) Planned rows of individual queries in Workload1



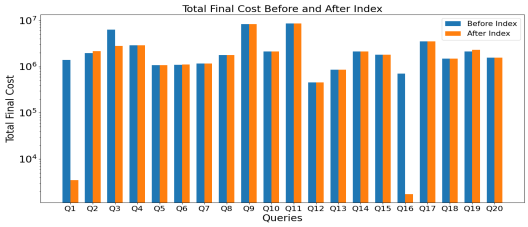
(b) Execution time of indv. queries in Workload1

Figure 8: Planned rows and Execution time of individual queries in workload1

The initial costs (shown in Fig. 9a) and final costs (shown in Fig. 9b) for Q1 and Q16 decreased sharply after creating indexes. The common point in these queries is the use of data ->> 'data-CNSSRFDataTypeName' with a NULL condition in the WHERE clause. There is a significant difference in performance between queries that select particular values from the indexed column and those that select NULL values from the indexed column in the query. The presence of an index allows the query planner to predict a lower cost for queries filtering on data ->> 'data-CNSSRFDataTypeName' is NULL. It can quickly locate the first row that meets the NULL condition, reducing the initial cost, and the index allows for faster retrieval of all matching rows with the NULL condition, resulting in a significant reduction in final costs as well. For Q3, while the execution time increased due to the overhead of handling DISTINCT operations, the costs related to row retrieval likely decreased as the index still provides quicker access to the relevant rows for aggregation, balancing out the increase in time with a decrease in raw computational cost.



(a) Initial costs of individual queries in Workload1



(b) Final costs of individual queries in Workload1

Figure 9: Initial costs and Final costs of individual queries in Workload1

The indexes likely made it easier for the database to identify and count NULL values for Q1 and Q16. However, due to the index maintenance or reorganization during the query, more rows might be processed internally, leading to an apparent increase in actual rows (shown in Fig. 10) counted. For Q2, Q3, Q10, and Q18, the common operation is counting the records on indexed columns. These queries benefit from the index by reducing the number of rows that need to be scanned or processed. The index provides direct access to the relevant data, thereby reducing the internal row count during the execution phase.

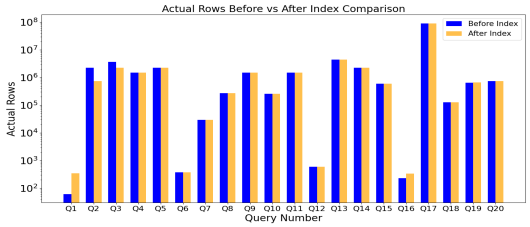


Figure 10: Actual rows of individual queries in Workload1

6.1.2 Workload 2:

It consists of 20 different queries, each employing a variety of query operators such as aggregate, distinct, like, date functions, null conditions, where, group by, order by clauses, and more. The purpose of these queries is to evaluate the effectiveness of different operators when used with an index, especially in terms of energy consumption. Fig. 11, represents the total cost and row values of the entire workload2 extracted from the PostgreSQL query plan before and after creating indexes on the workload2.

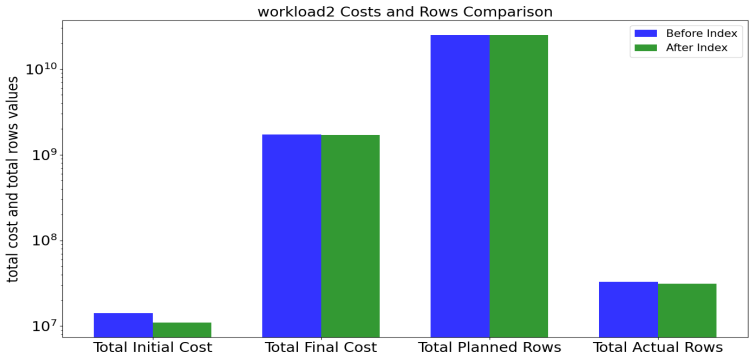
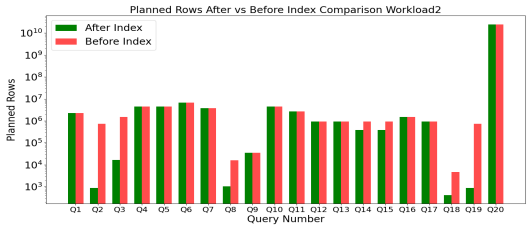
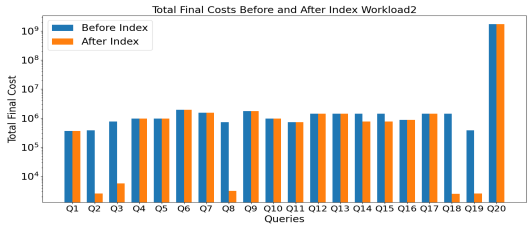


Figure 11: Workload 2 costs and rows comparison before and after creating indexes

After analyzing individual query results, we observed a significant decrease in the planned rows (shown in Fig. 12a) and final costs (shown in Fig. 12b)for specific queries [Q2, Q3, Q8, Q18, and Q19]. This reduction is attributed to the presence of an indexed column data ->> 'data-temperature' in the WHERE clause, which applies a specific condition across these queries.



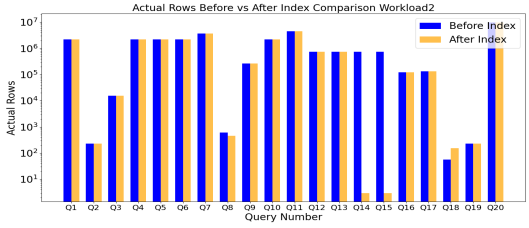
(a) Planned rows of individual queries in workload2



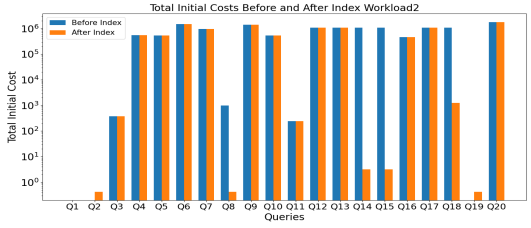
(b) Final costs of individual queries in workload2

Figure 12: Planned rows and Final costs of individual queries in workload2

The initial costs (shown in Fig. 13b) and actual rows (shown in Fig. 13a) were decreased for queries [Q14, Q15] sharply. These queries use aggregation functions (MIN and MAX) to retrieve statistical data of data ->> 'data-temperature' from the 'montoldre' dataset. These queries originally required a sequential scan (seq scan) of the entire dataset to compute the MIN and MAX temperature values. However, the index on data ->>'data-temperature' allows PostgreSQL to quickly locate and retrieve the maximum and minimum values without scanning all rows in the dataset sequentially. This optimization significantly reduces the computational resources and time required to execute these queries.



(a) Actual rows of individual queries in workload2

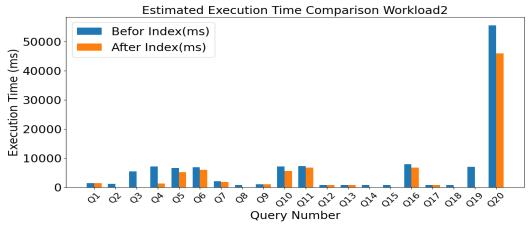


(b) Initial costs of individual queries in workload2

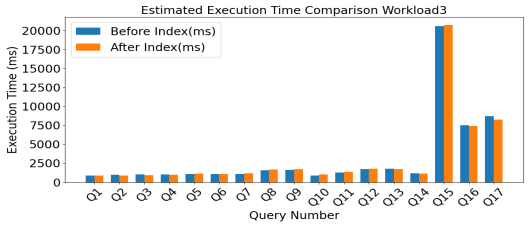
Figure 13: Actual rows and Initial costs of individual queries in workload2

Also, the initial costs (shown in Fig. 13b) decreased sharply for queries [Q8, Q18] because they could retrieve results directly from the index after its creation, optimizing the retrieval of data ->>'data-temperature' values within the specified range without needing to scan all rows sequentially resulting in reduced initial costs. On the other side, the initial costs increased slightly for queries [Q2, Q19], although the query uses the indexed data ->>'data-temperature' column in the WHERE clause, it also needs to scan the database to retrieve values from another column based on the specified condition which involves additional database scanning or operations beyond index usage, saw marginal increases in initial costs despite index utilization.

The implementation of an index on the data ->>'data-temperature' column in workload2 demonstrated substantial benefits in query execution time (shown in Fig. 14a) for most of the queries. By enabling efficient data access and retrieval, the index enhanced the execution speed of queries that leverage the indexed column, thereby optimizing overall database performance and responsiveness.



(a) Execution time of indv. queries in workload2



(b) Execution time of indv. queries in workload3

Figure 14: Execution times of individual queries in workload2 and workload3

6.1.3 Workload 3:

In sensor network databases, queries are frequently run concerning 'servertimestampUTC' to determine the different values received by sensors for various applications. This timestamp indicates the time when the data was received or processed by the server from the sensor node. It helps in tracking when the data arrives at the server, which can be useful for understanding network latency, server load times, or the data processing pipeline. Additionally, this index is currently utilized in the CEBA platform (based on the database structure I received from Jeremy). Therefore, we designed queries based on this criterion to evaluate how efficiently the data ->> 'servertimestampUTC' index benefits the database. It consists of 17 queries, the primary difference among these queries is the retrieval of different data type values in data ->> 'data-CNSSRFDataTypeName' for specific time intervals in data ->> 'servertimestampUTC'. Fig. 15 shows the values extracted from PostgreSQL before and after creating indexes on this workload.

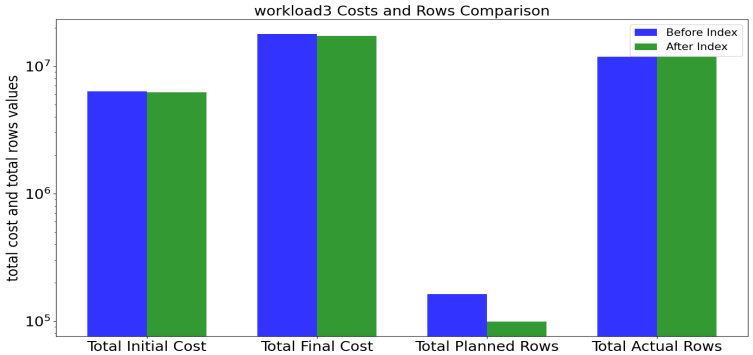


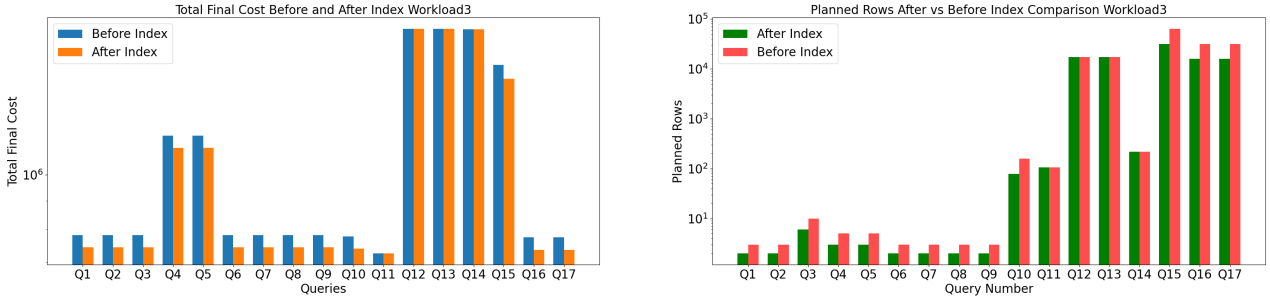
Figure 15: Workload 3 costs and rows comparison before and after creating Indexes

The data extracted from the data ->> 'servertimestampUTC' is not considered immutable because it represents the exact timestamp when data is received by the server and should remain constant in the database. However, the immutability constraint in PostgreSQL for index expressions requires that the function always returns the same result for the same input arguments, independent of external factors such as the current time, configuration settings, or timezone. Nevertheless, for better performance and

to ensure the index works efficiently, we created the index on the text representation of the timestamp and cast it to a timestamp data type in queries.

The execution time (shown in Fig. 14b)for most queries in workload3 increased slightly because of using the CAST operator to convert the text index to a timestamp within the queries. While creating an index on the text representation of a timestamp and casting it in queries is possible, this approach can negatively impact execution time due to inefficient index usage and the overhead of casting.

After creating an index on data `->>` 'servertimestampUTC' column, we observed that final costs (shown in Fig. 16b) and planned rows (shown in Fig. 16a) decreased slightly (which not considerable), indicating that the workload prefers sequential scans over index scans despite the index's presence in the database. Initial costs and actual row counts remained largely unchanged across all queries, with no significant differences observed. Therefore, we have not included these graph details in the presentation."



(a) Final costs of individual queries in workload3

(b) Planned rows of indv queries in workload3

Figure 16: Final costs and Planned rows of individual queries in workload3

7 Energy and Execution time

The 'powercfg/batteryreport' API is a command-line tool in Windows that generates a detailed report of the battery usage and energy consumption of the system. This report provides insights into the energy consumption patterns over a period. We use this API to measure the overall energy consumption of the system during the execution of various query plans. By analyzing the energy consumption data, we can assess the efficiency of different query plans in terms of power usage.

We measure the overall system energy consumption by running the 'powercfg/batteryreport' command automatically before running the first query in the workload and after executing the last query. This is done using a Python script, which provides a report with the power values for the entire system.

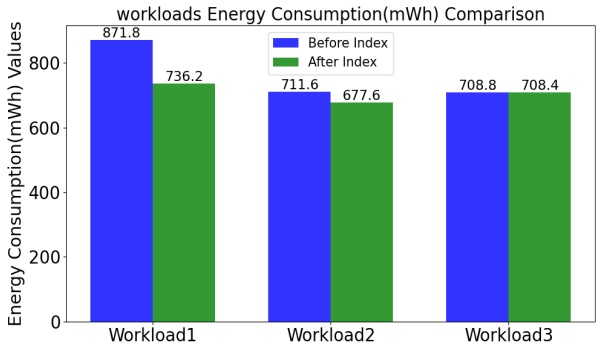
We measure the execution time of each query in the workload, the total execution time of the workload, and the average execution time of the workload by using the datetime module in the Python script during the workload execution.

7.1 Database Size: 2.5 GB

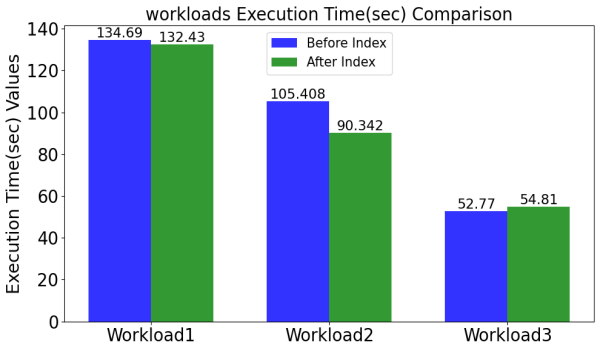
As mentioned previously, we ran each workload five times and represented the average energy and time consumption values for the overall workloads before and after creating indexes on the database. Fig. 17a, Fig. 17b represents average energy consumption and execution time for all three workloads at database size stands for 2.5 GB.

For the first two workloads, the indexes reduced both energy consumption and execution time. However, for workload 3, due to the use of the CAST function in queries to read data from data `->>` 'servertimestampUTC' as timestamps, the workload favored sequential scans over index scans even after index creation. This led to a very slight increase in execution time for workload3, while energy consumption remained consistent before and after indexing.

Fig. 18 represents the metrics that illustrate the differences in total initial costs, total final costs, total actual rows, and total planned rows for each workload before and after index implementation. As we see in the 'Max Diff Workload' column, workload1 exhibits the maximum differences in total initial cost and total actual rows before and after index implementation compared to the remaining two workloads, which indicates significant variations in actual energy consumption as we can see in Fig. 17a for workload1.



(a) Energy consumption comparison for Workloads



(b) Execution time comparison for Workloads

Figure 17: Overall energy consumption and execution time of workloads at database size of 2.5 GB

For workload2, the maximum difference values are observed in total final cost, and total planned rows before and after index usage compared to the other workloads. However, it's important to note that total planned rows are estimates and may not directly correlate with energy consumption. So, as it has a maximum difference in only one value indicates a minimal difference in energy consumption in workload2 as we can see in Fig. 17a for workload2.

In contrast, workload3 maintains consistent values before and after index implementation across all costs and rows. As we can see a negative difference in total actual rows suggests that the total no.of actual rows after the index is greater than before indexing with a value of 307 rows for the entire workload which is negligible, it's maybe because of the reorganization of the indexed column during the query, more rows might be processed internally, which aligns with the very minimal difference in energy consumption observed in Fig. 17a for workload3.

Regarding execution time, workload2 displays significant differences compared to the other workloads, reflecting more substantial variations in query plan execution. This finding is consistent with the higher execution time difference highlighted in Fig. 17b when compared to workloads 1 and 3.

We observe that both are proportional when comparing energy consumption differences with the metrics values. This suggests that the energy values measured by the 'powercfg' API command are accurate for the workloads, particularly since we cleared the cache before executing the workloads each time. Overall, these metrics emphasize the varying impacts of index implementation across different workloads and align with actual energy consumption and performance measured by our API.

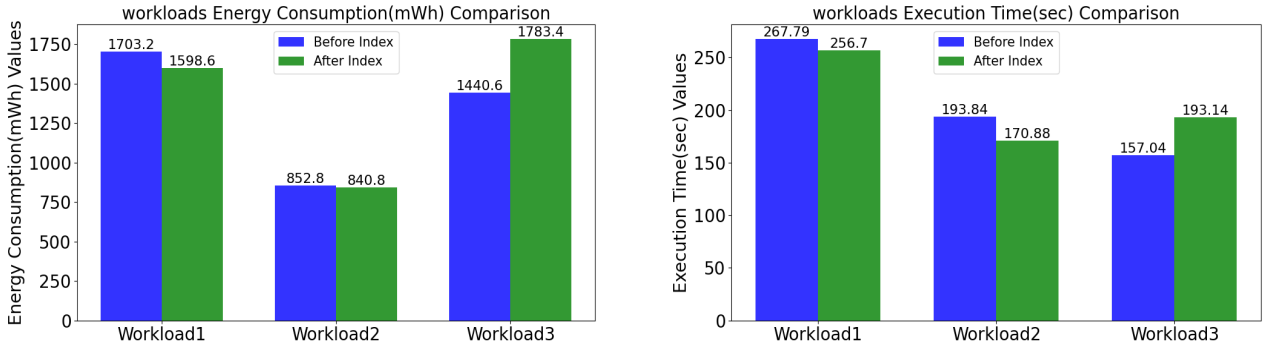
Cost and Rows				
Metric	**Workload 1**	**Workload 2**	**Workload 3**	**Max Diff Workload**
Total Initial Cost	4931572.10	3226837.20	101259.27	Workload 1
Total Final Cost	5198274.35	5969783.89	572591.39	Workload 2
Total Actual Rows	3996662	1490549	-307	Workload 1
Total Planned Rows	7985539	17996898	63161	Workload 2
Time				
Metric	**Workload 1**	**Workload 2**	**Workload 3**	**Max Diff Workload**
Total Exe Time	5500.589	38244.803	170.717	Workload 2

Figure 18: Cost, rows, and time difference metrics for all 3 workloads at database size of 2.5GB

7.2 Database Size: 4.5 GB

After completing all experiments on the 2.5 GB database, we added new records to increase the database size by 2 GB, resulting in a total size of 4.5 GB, We then continued our experiments on this expanded database. Fig. 19a, Fig. 19b represents the average energy consumption and execution time for all three workloads at a database size of 4.5 GB.

We observe that the differences in values before and after indexing are proportional to those obtained for the 2.5 GB database. For the first two workloads, the indexes reduced both energy consumption and execution time. However, for workload 3, due to the use of the CAST function in queries to read data from ->> 'servertimestampUTC' as timestamps, the workload favored sequential scans over index scans even after index creation. This led to an increase in both execution time and energy consumption for workload 3 after indexing.



(a) Energy consumption comparison for Workloads

(b) Execution time comparison for Workloads

Figure 19: Comparision of overall energy consumption and execution time of workloads at database size of 4.5 GB

Fig. 20 represents the metrics illustrate the differences in total initial costs, total final costs, total actual rows, and total planned rows for each workload before and after index implementation for a database size of 4.5 GB. As shown in the 'Max Diff Workload' column, workload 1 exhibits the maximum differences in total initial cost and total actual rows before and after index implementation compared to the other two workloads. This indicates significant variations in actual energy consumption, as seen in Fig. 19a for workload1.

Cost and Rows					
Metric	**Workload 1**	**Workload 2**	**Workload 3**	**Max Diff**	**Max Diff Workload**
Total Initial Cost	6550718.06	5227518.57	16.19	6550718.06	Workload 1
Total Final Cost	7390570.30	8212046.15	348.90	8212046.15	Workload 2
Total Actual Rows	5698526.00	2322146.00	-164.00	5698526.00	Workload 1
Total Planned Rows	12849811.00	4832102.00	28.00	12849811.00	Workload 1
Time					
Metric	**Workload 1**	**Workload 2**	**Workload 3**	**Max Diff**	**Max Diff Workload**
Total Exe Time	33185.759	42104.942	-613.37	42104.942	Workload 2

Figure 20: Cost, rows, and time difference metrics for all 3 workloads at database size of 4.5GB

For workload 2, the maximum difference values are observed in total final cost and total planned rows before and after index usage compared to the other workloads. However, it's important to note that total planned rows are estimates and may not directly correlate with energy consumption. Therefore, the maximum difference in only one value (total final cost) indicates a minimal difference in energy consumption for workload 2, as shown in Fig. 19a.

Regarding execution time, workload2 displays significant differences compared to the other workloads, reflecting more substantial variations in query plan execution. This finding is consistent with the higher execution time difference highlighted in Fig. 19b when compared to workloads 1 and 3.

% of optimization for each workload				
Parameter	DB Size	Workload1	Workload2	Workload3
Time	2.5 GB	1.68%	14.29%	-3.87%
	4.5 GB	4.14%	11.88%	-22.98%
Energy	2.5 GB	15.55%	4.78%	0.06%
	4.5 GB	6.13%	1.41%	-23.80%

Figure 21: % of optimization for each workload

Fig. 21 shows the percentage of optimization in time and energy for each workload at two different database sizes. We can see that, We can see that, Workload 2 benefits the most in terms of reduced execution time, with the highest percentage of optimization compared to other workloads. Workload 1 shows some improvement in execution time at both database sizes. Workload 3 experiences an increase in execution time, indicated by negative percentages, showing that the index actually results in worse performance for this workload.

Workload 1 shows significant improvement in energy consumption, particularly at the 2.5 GB database size. Workload 2 shows modest improvement in energy efficiency, with a more pronounced effect at the 2.5 GB size. Workload 3 experiences an increase in energy consumption, with negative percentages indicating a deterioration in energy efficiency, especially at the 4.5 GB size.

A key observation is that while using indexes can improve execution time, it does not necessarily reduce energy consumption proportionally. In our scenario, workload1 shows efficiency in reducing energy consumption with indexes but not in improving execution time. Conversely, workload2 benefits from improved execution time but does not optimize energy consumption effectively. Therefore, it is crucial to conduct a thorough analysis of both time and energy consumption improvements in the database before implementing and utilizing indexes.

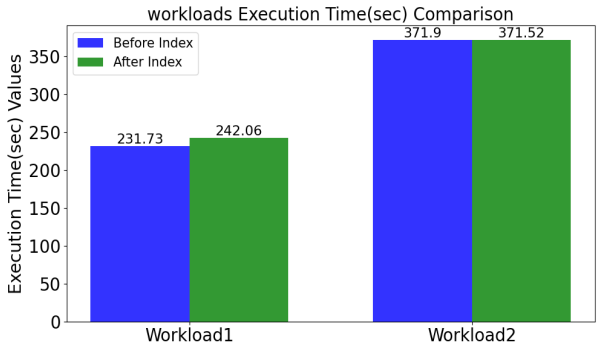
7.3 Write-Only Queries

It is widely recognized that indexes can introduce performance degradation in write-heavy scenarios. This makes our sequential approach crucial for evaluating both time and energy consumption for write-only queries with and without the use of indexes.

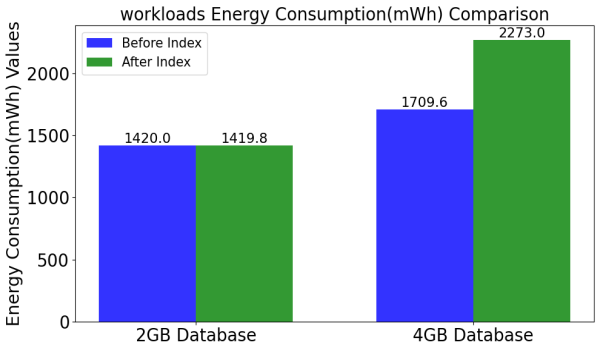
For our experiments, we used five update queries in a workload. Similar to the read-only query workloads, we analyzed the average values for the write-only query workload by running the workload five times each before and after creating indexes in the database.

Fig. 22b, Fig. 22a represents the average energy consumption and execution time for the write-only workload at database sizes of 2.5 GB and 4.5 GB. As shown in Fig. 22, the execution time for write-only queries is not affected by the presence of indexes. However, we observe a significant difference in energy consumption after indexing the 4.5 GB database. This increase in energy consumption might be attributed to the larger database size, which requires more energy to update both the database and the indexes as the size increases.

This finding highlights the importance of considering both execution time and energy consumption when evaluating the impact of indexes on write-heavy workloads. While indexes can optimize read performance, their effect on write operations, particularly in terms of energy consumption, can vary significantly depending on the database size and the nature of the queries.



(a) Energy consumption comparison for Workloads



(b) Execution time comparison for Workloads

Figure 22: Comparision of overall energy consumption and execution time of workloads at database size of 2.5 GB and 4.5 GB

We attempted to conduct similar experiments for INSERT and DELETE write operations to analyze execution time and energy consumption. However, the workloads related to these operations were executed very quickly, with a maximum execution time of less than 40 seconds, even when using the largest possible data record size available in my database. Due to the short duration of these operations, We were unable to accurately measure energy consumption using API.

Moreover, repeatedly inserting and deleting large volumes of data could potentially degrade the performance of my server. These limitations made it impractical to perform thorough experiments on INSERT and DELETE operations. Consequently, We did not include experiments for these operations in our analysis.

This limitation highlights a drawback in evaluating the energy consumption and execution time for short-lived write operations. Future work could explore alternative methods or tools better suited for measuring energy consumption in such scenarios, or consider scaling the database environment to better simulate more realistic, large-scale workloads.

8 Conclusion

The review of related work has provided valuable insights into existing methodologies aimed at optimizing energy consumption in databases. Our research introduces a novel approach to reducing energy consumption in database operations by using indexes. Our approach involves the development of an algorithm designed to efficiently extract query columns from training workloads. This algorithm then utilizes these inputs to generate optimal index configurations for the database, specifically aimed at minimizing energy usage during workload execution.

To evaluate the effectiveness of our approach, we designed and executed three distinct workloads consisting of 20, 20, and 17 queries respectively. These workloads were run both with and without the use of indexes to measure energy consumption and execution time. Our findings revealed that while indexes can significantly enhance performance in terms of execution time, the correlation with reduced energy consumption is not always proportional. Conversely, some configurations that effectively reduced energy consumption did not consistently optimize execution time.

In conclusion, our study underscores the importance of balancing performance enhancements with energy efficiency in database management. By systematically exploring index configurations and their impact on both energy consumption and execution time, our approach provides a practical framework for database administrators to make informed decisions toward sustainable and efficient index usage.

9 Future Work

The current phase of our work necessitates several improvements and future enhancements. Following the execution of experiments and the evaluation of various index configurations, a critical next step is to analyze the storage space requirements and maintenance overhead associated with each configuration. This analysis will provide valuable insights into the overall efficiency and sustainability of our chosen index configurations within real-world database environments

In addition, given that we are utilizing a commercial database and custom queries, adopting standard benchmark queries such as TPC becomes essential for a comparative analysis of index utilization across databases. These benchmarks offer standardized queries that are widely accepted across research communities, particularly in the field of database analysis and performance optimization.

Lastly, the use of power meters is indispensable for accurately measuring and analyzing both energy consumption and execution time of queries. These metrics serve as foundational data points for developing methodologies aimed at optimizing energy consumption in databases. Power meters play a pivotal role in ensuring precise measurements.

References

[1] Rajkumar Buyya, Anton Beloglazov, and Jemal Abawajy. Energy-efficient management of data center resources for cloud computing: A vision, architectural elements, and open challenges. *PDPTA*, 06 2010.

[2] Louise Krug, Mark Shackleton, and Fabrice Saffre. Understanding the environmental costs of fixed line networking. *e-Energy 2014 - Proceedings of the 5th ACM International Conference on Future Energy Systems*, 06 2014.

[3] Yongqiang Gao, Haibing Guan, Zhengwei Qi, Bin Wang, and Liang Liu. Quality of service aware power management for virtualized data centers. *Journal of Systems Architecture*, 59(4):245–259, 2013.

[4] Tiantian Li, Ge Yu, Xuebing Liu, and Jie Song. Analyzing the waiting energy consumption of nosql databases. In *2014 IEEE 12th International Conference on Dependable, Autonomic and Secure Computing*, pages 277–282. IEEE, 2014.

[5] E. Liebert. Five strategies for cutting data center energy costs through enhanced cooling efficiency, white paper. 2007.

[6] Dimitris Tsirogiannis, Stavros Harizopoulos, and Mehul Shah. Analyzing the energy efficiency of a database server. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 231–242, 06 2010.

[7] Amine Roukh, Ladjel Bellatreche, Selma Bouarar, and Ahcene Boukorca. Eco-physic: Eco-physical design initiative for very large databases. *Information Systems*, 68:44–63, 2017. Special issue on DOLAP 2015: Evolving data warehousing and OLAP cubes to big data analytics.

[8] Divya Mahajan, Cody Blakeney, and Ziliang Zong. Improving the energy efficiency of relational and nosql databases via query optimizations. *Sustainable Computing: Informatics and Systems*, 22:120–133, 2019.

[9] Ramakrishnan Kandhan Willis Lang and Jignesh M Patel. Rethinking query processing for energy efficiency: Slowing down to win the race. *IEEE Data Eng. Bull*, 34(1):12–23, 2011.

[10] Puneet K Birwa Mayuresh Kunjir and Jayant R Haritsa. Peak power plays in database engines. *In Proceedings of the 15th International Conference on Extending Database Technology*, page 444–455, 2012.

[11] Haijie Wang Xiaowei Liu, Jinbao Wang and Hong Gao. Generating power-efficient query execution plan. *In 2nd International Conference on Advances in Computer Science and Engineering (CSE 2013), Atlantis Press*, page 284–288, 2013.

[12] Bin Liao Dexian Yang Binglei Guo, Jiong Yu and Liang Lu. A green framework for dbms based on energy-aware query optimization and energy efficient query processing. *Journal of Network and Computer Applications*, 84:118–130, 2017.

[13] Jaime Seguel Manuel Rodriguez-Martinez, Harold Valdivia and Melvin Greer. Estimating power/energy consumption in database servers. *Procedia Computer Science*, 6:112–117, 2011.

[14] Yi-Cheng Tu Zichen Xu and Xiaorui Wang. Exploring power-performance tradeoffs in database systems. *In 2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, page 485–496, 2010.

[15] Zichen Y. Xu, Tu, and X. Wang. Dynamic energy estimation of query plans in database systems. *IEEE 33rd International Conference on Distributed Computing Systems*, page pp 83–92, 2013.

[16] Yi-Cheng Tu Zichen Xu and Xiaorui Wang. Online energy estimation of relational operations in database systems. *IEEE transactions on computers 64*, 11:pp 3223–3236, 2015.

[17] Nishi Gupta Nimesh Thakur. Relational and non relational databases: A review. *Journal of University of Shanghai for Science and Technology, ISSN: 1007-6735*, 23, Issue-8:pp 117–121, August - 2021.

[18] Bhat Uma and Jadhav Shraddha. Moving towards non-relational databases. *International Journal of Computer Applications*, 1, 02 2010.

[19] KASHYAP KUMAR, PANDEY B.K, Hardwari Mandoria, and KUMAR ASHOK. A review of leading databases: Relational and non-relational database. *i-manager's Journal on Information Technology*, 5:34, 01 2016.

[20] Konrad Fraczek and Malgorzata Plechawska-Wojcik. Comparative analysis of relational and non-relational databases in the context of performance in web applications. pages 153–164, 04 2017.

[21] Roman Čerešňák and Michal Kvet. Comparison of query performance in relational a non-relation databases. *Transportation Research Procedia*, 40:170–177, 2019. TRANSCOM 2019 13th International Scientific Conference on Sustainable, Modern and Safe Transport.

[22] D. Duarte and O. Belo. Evaluating query energy consumption in document stores. *in International Conference on Emerging Technologies for Developing Countries, Springer, 2017*, page pp 79–88, 2017.

[23] João Saraiva, Miguel Guimarales, and Orlando Belo. An economic energy approach for queries on data centers. *Universidade do Porto. Faculdade de Economia*, 2017.

[24] Miguel Guimarães, João Saraiva, and Orlando Belo. Some heuristic approaches for reducing energy consumption on database systems. 2016.

[25] Carlos Gomes, Eduardo Antonio Guimarães Tavares, and Meuse Nogueira de O. Junior. Energy consumption evaluation of nosql dbmss. *Anais do Workshop em Desempenho de Sistemas Computacionais e de Comunicação (WPerformance)*, 2020.

[26] Y. Tu Z. Xu and X. Wang. Pet: reducing database energy cost via query optimization. *Proceedings of the VLDB Endowment*, 5:pp 1954–1957, 2012.

[27] Balaji Subramaniam and Wu Feng. On the energy proportionality of distributed nosql data stores. volume 8966, pages 264–274, 04 2015.

[28] Surajit Chaudhuri and Vivek R. Narasayya. An efficient cost-driven index selection tool for microsoft sql server. In *Very Large Data Bases Conference*, 1997.

[29] Wendel Goes Pedrozo and Maria Salete Marcon Gomes Vaz. A tool for automatic index selection in database management systems. In *2014 International Symposium on Computer, Consumer and Control*, pages 1061–1064, 2014.

[30] Cecilia Cioloca, Mihai Georgescu, et al. Increasing database performance using indexes. *Database Systems Journal*, 2(2):13–22, 2011.