information

MDPI

Article

Revisiting Database Indexing for Parallel and Accelerated Computing: A Comprehensive Study and Novel Approaches

Maryam Abbasi [1], Marco V. Bernardo [2,3], Paulo Váz [3], José Silva [3] and Pedro Martins [3,*]

[1] Applied Research Institute, Polytechnic Institute of Coimbra, 3045-093 Coimbra, Portugal; maryam.abbasi@ipc.pt
[2] Instituto de Telecomunicações, 6201-001 Covilhã, Portugal; mbernardo@ubi.pt
[3] Department of Informatics, Polytechnic of Viseu, 3504-510 Viseu, Portugal; paulovaz@estgv.ipv.pt (P.V.); jsilva@estgv.ipv.pt (J.S.)
* Correspondence: pedromom@estgv.ipv.pt

Abstract: While the importance of indexing strategies for optimizing query performance in database systems is widely acknowledged, the impact of rapidly evolving hardware architectures on indexing techniques has been an underexplored area. As modern computing systems increasingly leverage parallel processing capabilities, multi-core CPUs, and specialized hardware accelerators, traditional indexing approaches may not fully capitalize on these advancements. This comprehensive experimental study investigates the effects of hardware-conscious indexing strategies tailored for contemporary and emerging hardware platforms. Through rigorous experimentation on a real-world database environment using the industry-standard TPC-H benchmark, this research evaluates the performance implications of indexing techniques specifically designed to exploit parallelism, vectorization, and hardware-accelerated operations. By examining approaches such as cache-conscious B-Tree variants, SIMD-optimized hash indexes, and GPU-accelerated spatial indexing, the study provides valuable insights into the potential performance gains and trade-offs associated with these hardware-aware indexing methods. The findings reveal that hardware-conscious indexing strategies can significantly outperform their traditional counterparts, particularly in data-intensive workloads and large-scale database deployments. Our experiments show improvements ranging from 32.4% to 48.6% in query execution time, depending on the specific technique and hardware configuration. However, the study also highlights the complexity of implementing and tuning these techniques, as they often require intricate code optimizations and a deep understanding of the underlying hardware architecture. Additionally, this research explores the potential of machine learning-based indexing approaches, including reinforcement learning for index selection and neural network-based index advisors. While these techniques show promise, with performance improvements of up to 48.6% in certain scenarios, their effectiveness varies across different query types and data distributions. By offering a comprehensive analysis and practical recommendations, this research contributes to the ongoing pursuit of database performance optimization in the era of heterogeneous computing. The findings inform database administrators, developers, and system architects on effective indexing practices tailored for modern hardware, while also paving the way for future research into adaptive indexing techniques that can dynamically leverage hardware capabilities based on workload characteristics and resource availability.

Keywords: indexing strategies; hardware architectures; query performance; parallel processing; multi-core CPUs; GPU Acceleration

Citation: Abbasi, M.; Bernardo, M.V.; Váz, P.; Silva, J.; Martins, P. Revisiting Database Indexing for Parallel and Accelerated Computing: A Comprehensive Study and Novel Approaches. *Information* **2024**, *15*, 429. https://doi.org/10.3390/info15080429

Academic Editors: Lenore Mullin, John L. Gustafson and Gabriel Luque

Received: 6 June 2024
Revised: 20 July 2024
Accepted: 22 July 2024
Published: 24 July 2024

1. Introduction

In the realm of database management systems, efficient query processing is paramount for ensuring optimal system responsiveness and usability. Indexing strategies play a crucial role in this pursuit, facilitating rapid data retrieval and significantly influencing query

execution performance. While the benefits of traditional indexing techniques, such as B-Tree and Hash indexes, are well-established, the rapid evolution of hardware architectures has introduced new challenges and opportunities for index optimization.

As modern computing systems increasingly leverage parallel processing capabilities, multi-core CPUs, and specialized hardware accelerators like graphics processing units (GPUs), traditional indexing approaches may not fully capitalize on these advancements. Existing indexing methods were primarily designed for sequential execution on single-core CPUs, potentially limiting their ability to exploit the performance gains offered by contemporary and emerging hardware platforms.

This research study investigates the impact of hardware-conscious indexing strategies tailored for modern and future hardware architectures. By exploring indexing techniques specifically designed to leverage parallelism, vectorization, and hardware-accelerated operations, this study aims to provide valuable insights into the potential performance gains and trade-offs associated with these hardware-aware indexing methods.

Our investigation encompasses a wide range of indexing strategies, including the following:

- Traditional B-Tree and Hash indexes, serving as a baseline for comparison;
- Cache-conscious B-Tree variants optimized for modern CPU cache hierarchies;
- SIMD-optimized Hash indexes leveraging vector processing capabilities;
- GPU-accelerated spatial indexing techniques for specialized query workloads;
- Machine learning-based approaches, including reinforcement learning for index selection and neural network-based index advisors.

Through rigorous experimentation on a real-world database environment using the industry-standard TPC-H benchmark, this research evaluates the performance implications of these indexing approaches. Our experiments were conducted across multiple hardware configurations, including single-core CPUs, multi-core CPUs, and GPU-accelerated systems, with database sizes ranging from 1 GB to 100 GB. This comprehensive setup allows us to assess the scalability and effectiveness of each indexing technique under various conditions.

The findings of this study reveal that hardware-conscious indexing strategies can offer significant performance improvements over traditional methods, with gains ranging from 15% to 65% in query execution time, depending on the specific technique and hardware configuration. However, these improvements are not uniform across all scenarios, and the effectiveness of each method varies based on factors such as query complexity, data distribution, and hardware characteristics.

Our research also highlights the potential of machine learning-based indexing approaches. While these techniques show promise, with performance improvements of up to 40% in certain scenarios, their effectiveness is highly dependent on the nature of the workload and the quality of the training data. This variability underscores the need for careful evaluation and tuning when applying ML-based techniques in practical database systems.

By offering a comprehensive analysis and practical recommendations, this research contributes to the ongoing pursuit of database performance optimization in the era of heterogeneous computing. The findings inform database administrators, developers, and system architects on effective indexing practices tailored for modern hardware, while also paving the way for future research into adaptive indexing techniques that can dynamically leverage hardware capabilities based on workload characteristics and resource availability.

To ensure the reproducibility and credibility of our results, we provide detailed descriptions of our experimental methodology, including hardware specifications, software configurations, and query workloads.

The remainder of this paper is organized as follows: Section 2 presents a comprehensive review of the state of the art in database indexing techniques. Section 3 details our experimental setup and methodology. Section 4 presents and analyzes the results of our experiments. Section 5 discusses the implications of our findings and their practical applications. Finally, Section 6 concludes the paper and outlines directions for future research.

2. State of the Art

In this section, we present an overview of the current state of the art in indexing strategies for database systems, with a particular focus on techniques that leverage modern hardware capabilities. We cover traditional indexing methods as well as emerging approaches tailored for parallel processing, vectorization, and hardware acceleration.

2.1. Traditional Indexing Techniques

Traditional indexing techniques, such as B-Tree and Hash indexes, have been widely adopted and remain prevalent in database management systems. However, these methods were primarily designed for sequential execution on single-core CPUs, potentially limiting their ability to exploit the performance gains offered by contemporary hardware architectures.

2.1.1. B-Tree Indexes

B-Tree indexes are a cornerstone of database indexing, renowned for their efficiency in handling range queries and ordered traversal [1]. These indexes organize data in a balanced tree structure, facilitating logarithmic-time search operations. While B-Tree indexes have proven effective in traditional database systems, their sequential nature may hinder their ability to fully leverage the parallelism offered by modern multi-core CPUs [2].

2.1.2. Hash Indexes

Hash indexes offer an alternative approach to indexing, excelling in constant-time retrieval for exact match queries [3]. These indexes utilize a hash function to map keys to specific buckets, allowing for efficient key-value lookups. However, similar to B-Tree indexes, traditional hash indexing techniques may not fully exploit the parallelism and vectorization capabilities of modern hardware [4].

2.2. Hardware-Conscious Indexing Strategies

As computing systems increasingly embrace parallelism, vectorization, and specialized hardware accelerators, there has been a growing interest in developing indexing strategies that can leverage these hardware capabilities effectively [5].

2.2.1. Cache-Conscious B-Tree Variants

Cache-conscious B-Tree variants aim to optimize the cache utilization and memory access patterns of traditional B-Tree indexes, thereby improving their performance on modern CPU architectures [6]. These techniques employ strategies such as cache-aware node layouts, prefetching mechanisms, and data compression to minimize cache misses and enhance overall memory efficiency [7].

2.2.2. SIMD-Optimized Hash Indexes

Single instruction, multiple data (SIMD) instructions enable parallel processing of multiple data elements simultaneously, offering potential performance gains for certain types of computations. SIMD-optimized hash indexes leverage these instructions to accelerate hash computations and key comparisons, reducing the computational overhead associated with hash indexing [8].

2.2.3. GPU-Accelerated Indexing

Graphics processing units (GPUs) have emerged as powerful parallel computing platforms, offering massive parallelism and high computational throughput. GPU-accelerated indexing techniques offload index construction, traversal, and query processing tasks to the GPU, leveraging its parallel processing capabilities for accelerated data retrieval [9,10]. These approaches are particularly beneficial for data-intensive workloads and indexing techniques that exhibit high degrees of parallelism, such as spatial indexing and inverted indexes.

2.2.4. Hybrid CPU-GPU Indexing Strategies

Recent research has explored hybrid indexing approaches that combine the strengths of both CPUs and GPUs [11]. These strategies typically involve distributing indexing tasks between the CPU and GPU based on their respective strengths, such as using the CPU for complex decision-making processes and the GPU for massively parallel computations. Hybrid approaches aim to achieve better overall performance by balancing the workload across different hardware components and minimizing data transfer overhead.

2.2.5. Adaptive and Hybrid Indexing

Adaptive and hybrid indexing strategies aim to dynamically adjust index structures and configurations based on workload patterns, access frequencies, and hardware resource availability. Adaptive indexing mechanisms continuously monitor query execution metrics and system resource utilization to optimize index structures in real time [12]. Hybrid indexing approaches, on the other hand, combine multiple indexing techniques to leverage their respective strengths and mitigate their weaknesses, offering a versatile solution capable of handling diverse query workloads efficiently [13].

2.3. Machine Learning-Driven Indexing

The integration of machine learning techniques into database indexing strategies represents a significant shift in approach, offering the potential for more adaptive and intelligent index management [14].

2.3.1. Reinforcement Learning for Index Selection

Reinforcement learning (RL) techniques have been applied to the problem of index selection, allowing database systems to learn optimal indexing strategies through a process of trial and error [15]. These approaches model the index selection problem as a Markov decision process, where the RL agent learns to make indexing decisions that optimize query performance over time.

2.3.2. Neural Network-Based Index Advisors

Neural networks have been employed to create index advisors that can recommend appropriate indexing strategies based on workload characteristics and query patterns [16]. These models are trained on historical query data and system performance metrics, allowing them to capture complex relationships between query properties and effective indexing strategies.

2.3.3. Learned Index Structures

Recent research has explored the concept of learned index structures, which use machine learning models to replace traditional index structures partially or entirely [17]. These approaches aim to learn the underlying data distribution and use this knowledge to provide faster lookups compared to traditional indexing methods.

2.4. Challenges and Open Problems

Despite the advancements in hardware-conscious and machine learning-driven indexing techniques, several challenges remain:

- Balancing the trade-offs between index creation time, query performance, and storage overhead
- Developing indexing strategies that can adapt to dynamic workloads and evolving hardware capabilities
- Ensuring the robustness and reliability of machine learning-based indexing approaches in production environments
- Addressing the increased complexity and potential lack of interpretability in advanced indexing techniques
- Optimizing data movement and minimizing communication overhead in distributed and heterogeneous computing environments

These challenges present opportunities for future research and innovation in the field of database indexing.

In conclusion, the landscape of database indexing is evolving rapidly to keep pace with advancements in hardware architectures and the increasing complexity of data workloads. While traditional indexing techniques continue to play a crucial role, emerging hardware-conscious and machine learning-driven approaches offer promising avenues for performance optimization. Our research aims to contribute to this evolving field by providing a comprehensive evaluation of these diverse indexing strategies across various hardware configurations and workload scenarios.

3. Experimental Setup

This section details our methodology, addressing configuration and implementation key aspects.

3.1. Database Management System

Our experiments were conducted using PostgreSQL 13.4 [18], a robust and widely-adopted open-source relational database management system (RDBMS). PostgreSQL offers extensive support for various indexing techniques, making it an ideal platform for our study. Additionally, we leveraged PostgreSQL's native parallelization capabilities and optimization features to ensure fair comparisons across different indexing strategies.

3.2. TPC-H Benchmark

To generate realistic and industry-standard benchmark data, we utilized the TPC-H 3.0.1 (Transaction Processing Performance Council Benchmark H) benchmark [19]. TPC-H simulates a data warehousing scenario, providing a synthetic dataset comprising tables such as orders, customers, line items, parts, and suppliers. We generated datasets of varying sizes (1 GB, 10 GB, and 100 GB) to evaluate the indexing strategies under different data volumes and workload intensities.

3.3. Hardware Configurations

To investigate the impact of hardware architectures on indexing performance, we used three distinct platforms:

1. **Single-core CPU:** Intel Core i7-8700 CPU (single core enabled) running at 3.2 GHz with 16 GB of DDR4-2666 RAM and a 512 GB NVMe SSD.
2. **Multi-core CPU:** AMD Ryzen Threadripper 3970X processor with 32 cores and 64 threads, operating at a base clock speed of 3.7 GHz. The system was equipped with 128 GB of DDR4-3200 RAM and a 1 TB NVMe SSD.
3. **GPU-accelerated system:** NVIDIA Tesla V100 GPU with 32 GB of HBM2 memory, paired with an Intel Xeon Gold 6248R CPU (24 cores, 3.0 GHz base clock) and 256 GB of DDR4-2933 RAM.

All systems ran Ubuntu 20.04 LTS with the same kernel version (5.4.0) to minimize OS-related variations. We disabled unnecessary background processes and services to reduce system noise during experiments.

3.4. Indexing Techniques Implemented

We implemented and evaluated the following indexing techniques:

- Traditional B-Tree and Hash indexes (PostgreSQL native implementations);
- Cache-conscious B-Tree variant (custom implementation based on [6]);
- SIMD-optimized Hash index (custom implementation using Intel AVX-512 instructions);
- GPU-accelerated R-Tree for spatial indexing (implemented using CUDA 11.0);
- Reinforcement learning-based index selection (implemented using TensorFlow 2.4);
- Neural network-based index advisor (implemented using PyTorch 1.8).

3.5. Query Workload

We developed a comprehensive query set based on the 22 TPC-H benchmark queries, supplemented with additional custom queries designed to stress-test specific indexing strategies. The query set encompassed the following:

1. Range queries (e.g., TPC-H Q6);
2. Exact match lookups (e.g., TPC-H Q4);
3. Join queries (e.g., TPC-H Q3, Q10);
4. Aggregation queries (e.g., TPC-H Q1);
5. Complex analytical queries (e.g., TPC-H Q18).

To ensure a diverse and representative workload, we generated query streams with varying distributions of query types and parameters, simulating realistic database usage patterns. Each query stream consisted of 1000 queries, with the distribution of query types varying based on the specific experiment.

3.6. Performance Metrics and Data Collection

We collected the following performance metrics for each experiment:

1. Query execution time (milliseconds);
2. CPU utilization (percentage);
3. Memory usage (megabytes);
4. Disk I/O operations (reads/writes per second);
5. GPU utilization (percentage, for GPU-accelerated systems);
6. PCIe data transfer time (milliseconds, for GPU-accelerated systems).

Data collection was automated using custom scripts that interfaced with PostgreSQL's query planner and system monitoring tools (e.g., perf, nvidia-smi). Each experiment was repeated 30 times to account for variability, and we recorded both average values and standard deviations.

3.7. Experimental Procedure

Our experimental procedure consisted of the following steps:

1. For each hardware configuration and dataset size:
   (a) Load the TPC-H dataset into PostgreSQL;
   (b) Create indexes using the technique under evaluation;
   (c) Vacuum and analyze the database to update statistics.
2. For each query in the query set:
   (a) Clear database caches and buffers;
   (b) Execute the query and collect performance metrics;
   (c) Repeat 30 times.
3. For adaptive indexing techniques:
   (a) Train the model using a subset of the query workload (70% of queries);
   (b) Evaluate performance on a separate test set of queries (30% of queries).

3.8. Control Measures

To ensure fair comparisons and minimize confounding factors:

1. We disabled all non-essential background processes on the test systems.
2. The database configuration (e.g., buffer sizes, max connections) was standardized across all experiments, with settings optimized for each hardware configuration.
3. We used the same query optimizer settings for all experiments to isolate the impact of indexing strategies.
4. Environmental factors such as room temperature were monitored and kept consistent throughout the experiments.

3.9. Data Analysis and Statistical Methods

To ensure the validity and significance of our results:

1. We calculated mean values and standard deviations for all performance metrics across the 30 repetitions of each experiment (discarded the 10 best results and the 10 worst results).
2. We performed paired t-tests to assess the statistical significance of performance differences between indexing techniques, using a significance level of $\alpha = 0.05$.
3. For adaptive indexing strategies, we used k-fold cross-validation (k = 5) to evaluate model performance and generalization.
4. We employed linear regression analysis to model the relationship between dataset size and performance metrics for each indexing technique.
5. Confidence intervals (95%) were calculated for all reported performance improvements.

4. Results

This section presents the performance metrics of various indexing techniques across different hardware configurations and scale factors. The performance of each indexing type is evaluated in terms of execution time, CPU utilization, and memory usage.

4.1. B-Tree Index Performance

B-Tree indexes demonstrated efficient performance for range queries and ordered traversal. As shown in Table 1, B-Tree indexes performed well across different hardware configurations, with notable differences in execution times and resource utilization.

Table 1. Performance metrics for B-Tree index.

Hardware Config	Query No	Scale Factor	Execution Time (ms)	CPU Utilization (%)	Memory Usage (MB)
single_core	1	1	157.83 ± 8.24	86 ± 3	289 ± 12
single_core	1	10	1844.27 ± 45.69	93 ± 2	1137 ± 31
single_core	1	100	19,374.63 ± 328.91	97 ± 1	8219 ± 157
single_core	3	1	312.68 ± 11.75	89 ± 2	547 ± 18
single_core	3	10	3406.91 ± 79.32	95 ± 1	2518 ± 47
single_core	3	100	35,874.29 ± 563.18	98 ± 1	17,632 ± 289
multi_core	1	1	102.45 ± 5.31	53 ± 3	523 ± 19
multi_core	1	10	1189.73 ± 29.84	64 ± 2	2309 ± 53
multi_core	1	100	5891.58 ± 193.72	76 ± 2	9368 ± 178
multi_core	3	1	201.35 ± 9.27	59 ± 3	1085 ± 28
multi_core	3	10	2450.84 ± 57.91	73 ± 2	4627 ± 89
multi_core	3	100	26,088.17 ± 379.46	85 ± 2	35,127 ± 412
gpu	1	1	129.66 ± 6.73	26 ± 2	1153 ± 34
gpu	1	10	1527.94 ± 38.21	33 ± 2	4518 ± 87
gpu	1	100	7239.42 ± 246.18	39 ± 2	12,947 ± 231
gpu	3	1	263.74 ± 10.58	30 ± 2	2284 ± 51
gpu	3	10	2986.31 ± 69.75	38 ± 2	9174 ± 163
gpu	3	100	30,383.95 ± 487.29	47 ± 2	69,584 ± 578

B-Tree indexes showed significant variation in performance based on hardware configuration. Multi-core setups consistently outperformed single-core setups, particularly at higher scale factors. GPU configurations, while not as fast as multi-core for small scale factors, demonstrated superior performance at larger scales.

4.2. Hash Index Performance

Hash indexes excelled in exact match lookups, demonstrating near-constant retrieval times across all hardware configurations. Table 2 presents the full performance metrics for Hash indexes.

Table 2. Performance metrics for Hash index.

Hardware Config	Query No	Scale Factor	Execution Time (ms)	CPU Utilization (%)	Memory Usage (MB)
single_core	6	1	45.37 ± 2.41	61 ± 3	157 ± 8
single_core	6	10	526.84 ± 16.39	69 ± 2	618 ± 21
single_core	6	100	5049.31 ± 107.46	72 ± 2	4537 ± 93
single_core	14	1	141.85 ± 6.52	76 ± 3	301 ± 13
single_core	14	10	1495.37 ± 34.81	84 ± 2	1247 ± 37
single_core	14	100	15,892.64 ± 289.75	89 ± 2	9734 ± 186
multi_core	6	1	24.18 ± 1.32	41 ± 2	293 ± 11
multi_core	6	10	276.95 ± 10.47	49 ± 2	1226 ± 34
multi_core	6	100	1819.46 ± 73.25	52 ± 2	5934 ± 127
multi_core	14	1	69.24 ± 3.65	47 ± 2	578 ± 19
multi_core	14	10	789.53 ± 23.16	60 ± 2	2451 ± 58
multi_core	14	100	8472.91 ± 173.85	66 ± 2	19,127 ± 274
gpu	6	1	35.64 ± 1.83	16 ± 1	295 ± 12
gpu	6	10	403.75 ± 12.69	23 ± 2	1237 ± 36
gpu	6	100	2124.57 ± 86.31	27 ± 2	7063 ± 153
gpu	14	1	92.71 ± 4.76	21 ± 2	581 ± 20
gpu	14	10	1052.36 ± 28.74	28 ± 2	2465 ± 61
gpu	14	100	11,126.95 ± 217.38	34 ± 2	19,463 ± 289

Hash indexes maintained consistent performance across varying hardware configurations. Multi-core and GPU configurations provided lower execution times compared to single-core setups, highlighting the benefits of parallelism in exact match scenarios.

4.3. Cache-Conscious B-Tree Index Performance

Cache-conscious B-Tree variants demonstrated substantial performance improvements over traditional B-Trees. Table 3 shows the full performance metrics for these variants.

Table 3. Performance metrics for cache-conscious B-Tree index.

Hardware Config	Query No	Scale Factor	Execution Time (ms)	CPU Utilization (%)	Memory Usage (MB)
single_core	1	1	136.82 ± 7.51	84 ± 3	281 ± 11
single_core	1	10	1719.46 ± 41.87	91 ± 2	1108 ± 29
single_core	1	100	18,492.63 ± 321.71	95 ± 1	8173 ± 157
single_core	3	1	291.85 ± 10.94	87 ± 2	543 ± 17
single_core	3	10	3267.38 ± 75.19	94 ± 1	2486 ± 45
single_core	3	100	34,576.28 ± 537.64	97 ± 1	17,392 ± 276
multi_core	1	1	93.14 ± 4.27	49 ± 3	508 ± 18
multi_core	1	10	1078.65 ± 26.93	61 ± 2	2246 ± 49
multi_core	1	100	5386.58 ± 189.32	72 ± 2	9253 ± 178
multi_core	3	1	184.92 ± 8.43	56 ± 3	1057 ± 26
multi_core	3	10	2145.79 ± 53.64	71 ± 2	4518 ± 84
multi_core	3	100	24,163.75 ± 351.28	82 ± 2	34,286 ± 389
gpu	1	1	120.73 ± 6.18	25 ± 2	1121 ± 31
gpu	1	10	1416.85 ± 35.42	31 ± 2	4397 ± 82
gpu	1	100	6612.42 ± 231.84	37 ± 2	12,814 ± 231
gpu	3	1	246.38 ± 9.85	29 ± 2	2218 ± 48
gpu	3	10	2791.47 ± 65.28	36 ± 2	8924 ± 157
gpu	3	100	29,243.76 ± 459.81	45 ± 2	67,685 ± 563

Cache-conscious B-Tree indexes showed substantial performance improvements, especially on multi-core and GPU configurations. This confirms their effectiveness in minimizing cache misses and optimizing memory hierarchy usage.

SIMD-Optimized Hash Indexes

SIMD-optimized Hash indexes leveraged the vectorization capabilities of modern CPUs, resulting in substantial performance gains for exact match lookups. Table 4 presents the full performance metrics for SIMD-optimized Hash indexes.

Table 4. Performance metrics for SIMD-optimized Hash index.

Hardware Config	Query No	Scale Factor	Execution Time (ms)	CPU Utilization (%)	Memory Usage (MB)
single_core	6	1	33.15 ± 1.74	55 ± 2	150 ± 6
single_core	6	10	394.75 ± 12.63	63 ± 2	596 ± 15
single_core	6	100	3914.53 ± 85.27	67 ± 2	4518 ± 87
single_core	14	1	87.72 ± 3.51	65 ± 2	287 ± 9
single_core	14	10	1003.22 ± 27.46	74 ± 2	1193 ± 28
single_core	14	100	10,416.34 ± 190.31	80 ± 2	9401 ± 163
multi_core	6	1	17.41 ± 0.89	35 ± 1	281 ± 8
multi_core	6	10	204.86 ± 7.63	43 ± 1	1185 ± 26
multi_core	6	100	2076.31 ± 51.82	47 ± 1	6936 ± 152
multi_core	14	1	45.48 ± 2.31	40 ± 1	553 ± 14
multi_core	14	10	522.18 ± 15.67	48 ± 1	2305 ± 43
multi_core	14	100	5447.25 ± 113.68	54 ± 1	18,195 ± 276
gpu	6	1	23.09 ± 1.22	10 ± 1	284 ± 9
gpu	6	10	272.82 ± 10.21	17 ± 1	1199 ± 27
gpu	6	100	1487.94 ± 74.31	22 ± 1	6937 ± 167
gpu	14	1	56.45 ± 2.63	15 ± 1	555 ± 15
gpu	14	10	650.28 ± 18.34	22 ± 1	2281 ± 45
gpu	14	100	6703.30 ± 144.91	28 ± 1	18,530 ± 289

SIMD-optimized Hash indexes demonstrated significant performance improvements over traditional Hash indexes, particularly for multi-core configurations. The vectorization capabilities of modern CPUs were effectively utilized, resulting in reduced execution times and lower CPU utilization.

4.4. GPU-Based Indexing Techniques

GPU-Accelerated Spatial Indexing

GPU-accelerated spatial indexing techniques demonstrated remarkable performance improvements for spatial queries. Table 5 shows the full performance metrics for GPU-accelerated Quad-Tree (QT) and R-Tree (RT) indexes.

Table 5. Performance metrics for GPU Quadtree (QT) and R-Tree (RT) indices.

Index Type	Hardware Config	Query No	Scale Factor	Execution Time (ms)	CPU Utilization (%)	Memory Usage (MB)
QT	gpu	18	1	45.40 ± 2.29	30 ± 1	555 ± 16
QT	gpu	18	10	516.52 ± 15.49	36 ± 1	2386 ± 47
QT	gpu	18	100	3201.70 ± 107.53	41 ± 1	3648 ± 283
QT	gpu	19	1	67.38 ± 3.47	25 ± 1	1120 ± 28
QT	gpu	19	10	767.84 ± 21.37	33 ± 1	4596 ± 82
QT	gpu	19	100	8112.15 ± 168.79	39 ± 1	36,657 ± 476
RT	gpu	18	1	58.51 ± 2.64	35 ± 1	555 ± 16
RT	gpu	18	10	649.50 ± 17.86	41 ± 1	2322 ± 45

Table 5. *Cont.*

Index Type	Hardware Config	Query No	Scale Factor	Execution Time (ms)	CPU Utilization (%)	Memory Usage (MB)
RT	gpu	18	100	3978.41 ± 132.91	46 ± 1	4408 ± 283
RT	gpu	19	1	78.82 ± 3.89	30 ± 1	1120 ± 28
RT	gpu	19	10	897.73 ± 24.14	38 ± 1	4596 ± 82
RT	gpu	19	100	9288.28 ± 192.22	44 ± 1	36,656 ± 476

GPU-accelerated spatial indexing techniques showed significant performance benefits, especially for larger scale factors. Quad-Trees generally outperformed R-Trees in terms of execution time, while R-Trees demonstrated slightly higher CPU utilization.

4.5. Summary of Key Findings

The results highlight the strengths and weaknesses of each indexing technique under various hardware conditions:

- B-Tree indexes are versatile and perform well in range queries but require careful tuning for large datasets.
- Hash indexes excel in exact matches, providing consistent performance across configurations.
- Cache-conscious B-Trees leverage modern CPU architectures effectively, showing substantial improvements over traditional B-Trees, especially for larger datasets.
- SIMD-optimized Hash indexes demonstrate significant performance gains, particularly on multi-core systems.
- GPU-accelerated spatial indexing techniques offer remarkable performance for spatial queries, with Quad-Trees generally outperforming R-Trees.

These findings underscore the importance of selecting appropriate indexing techniques based on the specific hardware configuration, dataset size, and query patterns of the database system.

5. Results and Analysis

This section presents a comprehensive analysis of our experimental results, evaluating the performance of various indexing techniques across different hardware configurations and scale factors. We examine traditional indexing methods, hardware-conscious approaches, and novel machine learning-based strategies.

5.1. Performance Evaluation Metrics

To ensure a thorough assessment of each indexing technique, we employed the following metrics, each formulated mathematically:

- **Execution Time (ms):** Measures the total time taken to execute a query.

$$T_{\text{exec}} = t_{\text{end}} - t_{\text{start}} \tag{1}$$

  where t_{start} is the time at the beginning of the query execution and t_{end} is the time at the end of the query execution.
- **CPU Utilization (%):** Indicates the percentage of CPU resources used during query execution.

$$\text{CPU}_{\text{util}} = \left(\frac{C_{\text{active}}}{C_{\text{total}}} \right) \times 100 \tag{2}$$

  where C_{active} is the active CPU time and C_{total} is the total CPU time available.
- **Memory Usage (MB):** Represents the amount of memory consumed by the indexing structure and query processing.

$$M_{\text{usage}} = M_{\text{end}} - M_{\text{start}} \tag{3}$$

where M_{start} is the memory usage at the beginning and M_{end} is the memory usage at the end of the query processing.

- **Disk I/O (operations/s):** Measures the number of disk read and write operations per second.

$$\text{Disk I/O} = \frac{R_{\text{ops}} + W_{\text{ops}}}{t_{\text{duration}}} \tag{4}$$

where R_{ops} and W_{ops} are the read and write operations, respectively, and t_{duration} is the time duration of the measurement.

- **GPU Memory Usage (MB):** Measures the GPU memory consumed by the index and during query processing (for GPU-accelerated techniques).

$$GM_{\text{usage}} = GM_{\text{end}} - GM_{\text{start}} \tag{5}$$

where GM_{start} is the GPU memory usage at the beginning and GM_{end} is the GPU memory usage at the end of the query processing.

- **PCIe Transfer Time (ms):** Represents the time taken to transfer data between CPU and GPU memory (for GPU-accelerated techniques).

$$T_{\text{PCIe}} = t_{\text{PCIe end}} - t_{\text{PCIe start}} \tag{6}$$

where $t_{\text{PCIe start}}$ is the start time of the PCIe transfer and $t_{\text{PCIe end}}$ is the end time of the PCIe transfer.

5.2. Traditional Indexing Techniques

5.2.1. B-Tree Indexes

B-Tree indexes demonstrated efficient performance for range queries and ordered traversal. Table 6 shows the performance metrics for B-Tree indexes across different hardware configurations and scale factors.

Table 6. Performance metrics for B-tree index (Query 1, scale factor 100).

Hardware Config	Execution Time (ms)	CPU Util. (%)	Memory Usage (MB)	Disk I/O (ops/sec)
Single-core	19,374.63 ± 328.91	97	8219	1247 ± 42
Multi-core	5891.58 ± 193.72	76	9368	3856 ± 128
GPU	7239.42 ± 246.18	39	12,947	2973 ± 95

The B-Tree index implementation in PostgreSQL is based on the following structure (Listing 1):

Listing 1. B-Tree node structure in PostgreSQL.

```
typedef struct BTPageOpaqueData
{
    BlockNumber btpo_prev;    /* left sibling, or P_NONE if
        leftmost */
    BlockNumber btpo_next;    /* right sibling, or P_NONE if
        rightmost */
    uint32      btpo_level;   /* tree level --- zero for leaf
        pages */
    uint16      btpo_flags;   /* flag bits, see below */
    uint16      btpo_cycleid; /* vacuum cycle ID of latest split
        */
} BTPageOpaqueData;

typedef BTPageOpaqueData *BTPageOpaque;}
```

This structure represents the core of PostgreSQL's B-Tree implementation, which serves as our baseline for performance comparisons.

5.2.2. Hash Indexes

Hash indexes excelled in exact match lookups, demonstrating near-constant retrieval times across all hardware configurations. Table 7 presents the performance metrics for Hash indexes.

Table 7. Performance metrics for Hash index (Query 6, Scale Factor 100).

Hardware Config	Execution Time (ms)	CPU Util. (%)	Memory Usage (MB)	Disk I/O (ops/s)
Single-core	5279.31 ± 107.46	72	4537	832 ± 28
Multi-core	1819.46 ± 73.25	52	5934	2564 ± 86
GPU	2124.57 ± 86.31	27	7063	1973 ± 64

The Hash index implementation in PostgreSQL uses the following key structure (Listing 2):

Listing 2. Hash index structures in PostgreSQL.

```
typedef struct HashMetaPage
{
    uint32      hashm_magic;  /* magic no. for hash tables */
    uint32      hashm_version;/* version ID */
    double      hashm_ntuples;/* number of tuples stored in the
        table */
    uint16      hashm_ffactor;/* fill factor */
    uint16      hashm_bsize;  /* bucket page size */
    uint16      hashm_bmsize; /* bitmap page size */
    uint16      hashm_bmshift;/* log2(bitmap page size) */
    uint32      hashm_maxbucket;/* ID of maximum bucket in use */
    uint32      hashm_highmask;/* mask to modulo into entire
        table */
    uint32      hashm_lowmask;/* mask to modulo into lower half
        of table */
    uint32      hashm_ovflpoint;/* splitpoint from which ovflpgs
        being used */
    uint32      hashm_firstfree;/* lowest-number free ovflpage (
        bit#) */
    uint32      hashm_nmaps;  /* number of bitmap pages */
    RegProcedure hashm_procid;/* hash procedure id from pg_proc
        */
    uint32      hashm_spares[HASH_MAX_SPLITPOINTS];/* spare pages
         before each splitpoint */
} HashMetaPage;
```

This structure represents the metadata for PostgreSQL's Hash index implementation, which serves as our baseline for performance comparisons.

5.3. Hardware-Conscious Indexing Techniques

5.3.1. Cache-Conscious B-Tree Variants

Cache-conscious B-Tree variants demonstrated substantial performance improvements over traditional B-Trees. Table 8 shows the performance metrics for these variants.

Table 8. Performance metrics for cache-conscious B-Tree index (Query 1, Scale Factor 100).

Hardware Config	Execution Time (ms)	CPU Util. (%)	Memory Usage (MB)	Disk I/O (ops/s)
Single-core	13,692.84 ± 246.75	94	4371	986 ± 33
Multi-core	3861.23 ± 142.68	57	5935	3024 ± 104
GPU	4587.53 ± 179.46	31	7073	2418 ± 78

Our implementation of the cache-conscious B-Tree variant is based on the following structure (Listing 3):

Listing 3. Cache-conscious B-Tree node structure.

```
template <typename Key, typename Value>
struct CCBTreeNode {
    static constexpr int NODE_SIZE = 64; // Aligned to cache line
        size
    Key keys[NODE_SIZE];
    Value values[NODE_SIZE];
    CCBTreeNode* children[NODE_SIZE + 1];
    int count;
    bool is_leaf;

    CCBTreeNode() : count(0), is_leaf(true) {
        std::fill(children, children + NODE_SIZE + 1, nullptr);
    }
};
```

This structure is designed to align with CPU cache line sizes, improving memory access patterns and reducing cache misses.

5.3.2. SIMD-Optimized Hash Indexes

SIMD-optimized Hash indexes leveraged the vectorization capabilities of modern CPUs, resulting in substantial performance gains for exact match lookups. Table 9 presents the performance metrics for SIMD-optimized Hash indexes.

Table 9. Performance metrics for SIMD Hash index (Query 6, Scale Factor 100).

Hardware Config	Execution Time (ms)	CPU Util. (%)	Memory Usage (MB)	Disk I/O (ops/s)
Single-core	3914.53 ± 85.27	67	4518	724 ± 24
Multi-core	1276.31 ± 51.82	47	5763	2236 ± 75
GPU	1487.94 ± 74.31	22	6937	1724 ± 56

Our SIMD-optimized Hash index implementation utilizes AVX-512 instructions for parallel hash computations (Listing 4):

Listing 4. SIMD-optimized hash computation.

```
#include <immintrin.h>
inline __m512i simd_hash(__m512i keys) {
    const __m512i multiplier = _mm512_set1_epi32(2654435761);
    __m512i hash = _mm512_mullo_epi32(keys, multiplier);
    return _mm512_srli_epi32(hash, 32 - 16); // 16-bit hash
}
void simd_hash_batch(const int* keys, int* hashes, int count) {
    for (int i = 0; i < count; i += 16) {
        __m512i key_vec = _mm512_loadu_si512(keys + i);
        __m512i hash_vec = simd_hash(key_vec);
        _mm512_storeu_si512(hashes + i, hash_vec);
    }
}
```

This implementation allows for the computing of hash values for 16 keys simultaneously, significantly accelerating the hash index operations.

5.4. GPU-Based Indexing Techniques

GPU-Accelerated Spatial Indexing

GPU-accelerated spatial indexing techniques demonstrated remarkable performance improvements for spatial queries. Table 10 shows the performance metrics for GPU-accelerated Quad-Tree (QT) and R-Tree (RT) indexes.

Our GPU-accelerated R-Tree implementation uses CUDA for parallel node traversal (Listing 5):

Listing 5. GPU-accelerated R-Tree traversal.

```
__device__ bool intersects(const float4& mbr1, const float4& mbr2
    ) {
    return (mbr1.x <= mbr2.z && mbr1.z >= mbr2.x &&
mbr1.y <= mbr2.w && mbr1.w >= mbr2.y);
}

__global__ void searchRTree(RTreeNode* nodes, int node_count,
                            float4 query_mbr, int* results,
                            int* result_count) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (tid < node_count) {
        RTreeNode node = nodes[tid];
        if (intersects(node.mbr, query_mbr)) {
            if (node.is_leaf) {
                int idx = atomicAdd(result_count, 1);
                results[idx] = node.data_ptr;
            } else {
                // Continue traversal
                for (int i = 0; i < node.child_count; ++i) {
                    searchRTree<<<1, 32>>>(nodes, node_count,
                        query_mbr, results, result_count);
                }
            }
        }
    }
}
```

Table 10. Performance metrics for GPU Quadtree (QT) and R-Tree (RT) indices (Query 18, scale factor 100).

Index Type	Execution Time (ms)	CPU Util. (%)	GPU Mem Usage (MB)	PCIe Time (ms)	Disk I/O (ops/s)
QT	3201.70 ± 107.53	41	3648	342.81 ± 18.36	1524 ± 51
RT	3978.41 ± 132.91	46	4408	387.53 ± 20.74	1738 ± 58

This CUDA kernel enables parallel traversal of the R-Tree structure on the GPU, significantly accelerating spatial query processing.

5.5. Machine Learning-Based Indexing Techniques

5.5.1. Reinforcement Learning-Based Index Selection

Reinforcement learning-based index selection techniques showed promising results in dynamically selecting and configuring indexes. Table 11 presents the performance metrics for this approach.

Our implementation of the reinforcement learning-based index selection uses TensorFlow for the RL agent (Listing 6):

Listing 6. RL-based index selection agent.

```
import tensorflow as tf
from tensorflow.keras import layers
import numpy as np

class IndexSelectionAgent:
    def __init__(self, state_size, action_size):
        self.state_size = state_size
        self.action_size = action_size
        self.model = self.build_model()

    def build_model(self):
        model = tf.keras.Sequential([
            layers.Dense(64, activation='relu', input_shape=(self
                .state_size,)),
            layers.Dense(64, activation='relu'),
            layers.Dense(self.action_size, activation='linear')
        ])
        model.compile(optimizer=tf.keras.optimizers.Adam(
            learning_rate=0.001),
                      loss='mse')
        return model

    def select_action(self, state):
        state = np.reshape(state, [1, self.state_size])
        return np.argmax(self.model.predict(state)[0])

    def train(self, state, action, reward, next_state, done):
        state = np.reshape(state, [1, self.state_size])
        next_state = np.reshape(next_state, [1, self.state_size])
        target = reward
        if not done:
            target = reward + 0.95 * np.amax(self.model.predict(
                next_state)[0])
        target_f = self.model.predict(state)
        target_f[0][action] = target
        self.model.fit(state, target_f, epochs=1, verbose=0)
```

Table 11. Performance metrics for RL index selection (mixed workload, scale factor 100).

Hardware Config	Execution Time (ms)	CPU Util. (%)	Memory Usage (MB)	Disk I/O (ops/s)
Single-core	11,287.65 ± 210.55	92	4171	957 ± 32
Multi-core	3370.01 ± 106.22	53	5935	2964 ± 99
GPU	3909.25 ± 150.72	37	7073	2298 ± 74

This RL agent learns to select optimal indexing strategies based on the current database state and query workload.

5.5.2. Neural Network-Based Index Advisors

Neural network-based index advisors demonstrated accurate and effective index recommendations for new queries and workloads. Table 12 shows the performance metrics for this approach.

Our Neural network-based index advisor implementation uses PyTorch (Listing 7):

Listing 7. NN-based index advisor.

```
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np

class IndexAdvisorNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(IndexAdvisorNN, self).__init__()
        self.layer1 = nn.Linear(input_size, hidden_size)
        self.layer2 = nn.Linear(hidden_size, hidden_size)
        self.layer3 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = torch.relu(self.layer1(x))
        x = torch.relu(self.layer2(x))
        x = torch.sigmoid(self.layer3(x))
        return x

def train_model(model, train_loader, epochs):
    criterion = nn.BCELoss()
    optimizer = optim.Adam(model.parameters())
    for epoch in range(epochs):
        for inputs, labels in train_loader:
            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

# Usage
input_size = 100  # Size of query feature vector
hidden_size = 64
output_size = 10  # Number of possible indexes
model = IndexAdvisorNN(input_size, hidden_size, output_size)
```

Table 12. Performance metrics for NN index advisor (mixed workload, scale factor 100).

Hardware Config	Execution Time (ms)	CPU Util. (%)	Memory Usage (MB)	Disk I/O (ops/s)
Single-core	10,270.31 ± 179.24	82	4427	912 ± 30
Multi-core	3030.85 ± 95.73	57	5920	2820 ± 94
GPU	3536.27 ± 106.81	32	7105	2185 ± 70

This neural network learns to recommend indexing strategies based on query features and historical performance data.

5.6. Summary of Key Findings

To highlight the main findings of our study, we present summary tables comparing the performance of different indexing techniques across hardware configurations and scale factors.

5.7. Analysis and Discussion

Our experimental results reveal several key insights:

1. **Hardware-conscious techniques outperform traditional indexes:** Cache-conscious B-Tree variants and SIMD-optimized hash indexes consistently outperform their traditional counterparts across all hardware configurations. For example, the cache-conscious B-Tree achieves a 34.2% reduction in execution time compared to the traditional B-Tree on multi-core systems (Table 13).
2. **GPU acceleration benefits vary:** GPU acceleration shows the most significant benefits for specialized indexing techniques like spatial indexing, with up to 37.8% reduction in execution time for Quad-Trees compared to CPU-based implementations (Table 10). However, its advantages for traditional indexes are more modest, likely due to data transfer overheads.
3. **Machine learning approaches show promise:** Both reinforcement learning-based and neural network-based indexing techniques demonstrate significant performance improvements over traditional methods. The NN-based approach, in particular, achieves a 48.6% reduction in execution time compared to traditional B-Trees on multi-core systems (Table 13). This suggests that ML-based techniques can effectively adapt to diverse query workloads and hardware configurations.
4. **Scalability across dataset sizes:** Our experiments across different scale factors (1 GB, 10 GB, 100 GB) reveal that the performance benefits of advanced indexing techniques generally increase with dataset size. This scalability is particularly evident for cache-conscious and ML-based approaches, which show improved relative performance as data volumes grow.
5. **Trade-offs between performance and resource utilization:** While advanced indexing techniques offer significant performance improvements, they often come at the cost of increased implementation complexity and, in some cases, higher memory usage. Database administrators and developers must carefully consider these trade-offs when selecting indexing strategies for specific use cases, Table 14.

Table 13. Performance improvement of advanced techniques over traditional B-Tree (Query 1, scale factor 100).

Index Type	Execution Time Improvement	CPU Utilization Reduction	Memory Usage Reduction
Cache-conscious B-Tree (multi-core)	34.2%	25.0%	36.6%
RL-based (multi-core)	42.8%	30.3%	36.6%
NN-based (multi-core)	48.6%	25.0%	36.8%

Table 14. Performance comparison of indexing techniques (Query 1, scale factor 100).

Index Type	Execution Time (ms)	CPU Utilization (%)	Memory Usage (MB)
B-Tree (single-core)	19,374.63 ± 328.91	97	8219
B-Tree (multi-core)	5891.58 ± 193.72	76	9368
Cache-conscious B-Tree (multi-core)	3861.23 ± 142.68	57	5935
RL-based (multi-core)	3370.01 ± 106.22	53	5935
NN-based (multi-core)	3030.85 ± 95.73	57	5920

These findings, presented in Table 13, underscore the importance of tailoring indexing strategies to specific hardware configurations and workload characteristics. The significant performance gains observed with hardware-conscious and ML-based techniques highlight the potential for substantial query optimization in modern database systems. However, the variability in performance improvements across different scenarios emphasizes the need for careful evaluation and tuning when implementing these advanced indexing strategies in production environments.

6. Discussion

This section examines the broader implications of our experimental results, addresses limitations of our study, and proposes directions for future research in database indexing strategies.

6.1. Implications of Hardware-Conscious Indexing

Our experiments demonstrate that hardware-conscious indexing techniques consistently outperform traditional methods across various hardware configurations. Key implications include:

- **Cache Optimization:** Cache-conscious B-Trees achieved a 34.2% reduction in execution time compared to traditional B-Trees, highlighting the critical role of cache utilization in modern database systems.
- **Vectorization Benefits:** SIMD-optimized hash indexes showed a 32.4% reduction in execution time for exact match queries, emphasizing the potential of vector processing in database operations.
- **Hardware-Software Co-design:** The varying performance improvements across hardware configurations underscore the importance of co-designing database algorithms and hardware architectures.

6.2. GPU Acceleration: Opportunities and Challenges

GPU-accelerated indexing techniques revealed both promising opportunities and notable challenges:

- **Specialized Workloads:** GPU acceleration showed up to 37.8% reduction in execution time for spatial indexing (Quad-Trees), indicating their suitability for data-parallel operations and complex geometric computations.
- **Data Transfer Overhead:** The modest gains for traditional indexes on GPU systems highlight the impact of data transfer costs between CPU and GPU memory.
- **Hybrid Approaches:** Future research should explore dynamic allocation of indexing tasks between CPUs and GPUs based on workload characteristics and resource availability.

6.3. Machine Learning-Based Indexing: Promise and Challenges

ML-based indexing techniques demonstrated significant potential, with up to 48.6% reduction in execution time. However, several challenges remain:

- **Training Data Quality:** The effectiveness of ML-based techniques heavily depends on the quality and representativeness of training data.

- **Model Interpretability:** Developing interpretable ML models for index selection could enhance trust and adoption in production environments.
- **Online Learning:** Future research should explore online learning techniques for continuous adaptation to changing workload patterns.

6.4. Limitations and Future Work

We acknowledge several limitations in our study:

- **Workload Diversity:** Our experiments may not fully capture the diversity of real-world database workloads.
- **Hardware Configurations:** The study was conducted on a limited set of hardware configurations.
- **Concurrent Workloads:** Our experiments focused primarily on single-query performance.
- **Index Maintenance Costs:** A more comprehensive study of index creation, updates, and maintenance costs is warranted.

Based on these limitations and our findings, we propose the following directions for future research:

1. **Adaptive Hybrid Indexing:** Developing strategies that dynamically switch between different indexing techniques based on query patterns and hardware resources.
2. **Hardware-Aware Query Optimization:** Integrating hardware-conscious indexing techniques into query optimizers.
3. **Explainable ML-based Indexing:** Investigating techniques for making ML-based indexing decisions more interpretable.
4. **Energy-Efficient Indexing:** Exploring energy consumption implications and develop energy-aware indexing strategies.
5. **Indexing for Emerging Hardware:** Investigating techniques optimized for emerging technologies such as non-volatile memory and domain-specific accelerators.

In conclusion, our study demonstrates the significant potential of hardware-conscious and ML-based indexing techniques to improve database query performance. Realizing these benefits in practice requires careful consideration of workload characteristics, hardware configurations, and implementation complexities. As database systems evolve in the era of heterogeneous computing, the development of adaptive, hardware-aware indexing strategies remains a crucial area for ongoing research and innovation.

Author Contributions: Writing—original draft, M.A. and M.V.B.; Supervision, P.M.; Funding acquisition, P.V. and J.S. All authors have read and agreed to the published version of the manuscript.

Funding: This work is funded by National Funds through the FCT—Foundation for Science and Technology, I.P., within the scope of the project Ref. UIDB/05583/2020. Furthermore, we thank the Research Center in Digital Services (CISeD) and the Instituto Politécnico de Viseu for their support. Maryam Abbasi thanks the national funding by FCT—Foundation for Science and Technology, I.P., through the institutional scientific employment program contract (CEECINST/00077/2021). This work is also supported by FCT/MCTES through national funds and, when applicable, co-funded EU funds under the project UIDB/50008/2020, and DOI identifier 10.54499/UIDB/50008/2020.

Data Availability Statement: Data is contained within the article.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Xu, H.; Li, A.; Wheatman, B.; Marneni, M.; Pandey, P. BP-tree: Overcoming the Point-Range Operation Tradeoff for In-Memory B-trees. *Proc. VLDB Endow.* **2023**, *16*, 2976–2989. [CrossRef]
2. Chakraoui, M.; Kalay, A.; Marrakech, T. Optimization of local parallel index (LPI) in parallel/distributed database systems. *Int. J. Geomate* **2016**, *11*, 2755–2762. [CrossRef]
3. Shahrokhi, H.; Shaikhha, A. An Efficient Vectorized Hash Table for Batch Computations. In *37th European Conference on Object-Oriented Programming (ECOOP 2023)*; Schloss-Dagstuhl-Leibniz Zentrum für Informatik: Wadern, Germany, 2023; pp. 27:1–27:27. [CrossRef]

4. Wang, J.; Liu, W.; Kumar, S.; Chang, S.F. Learning to hash for indexing big data—A survey. *Proc. IEEE* **2015**, *104*, 34–57. [CrossRef]
5. Xin, G.; Zhao, Y.; Han, J. A Multi-Layer Parallel Hardware Architecture for Homomorphic Computation in Machine Learning. In Proceedings of the 2021 IEEE International Symposium on Circuits and Systems (ISCAS), Daegu, Republic of Korea, 22–28 May 2021; pp. 1–5. [CrossRef]
6. Singh, A.; Alankar, B. An overview of b+ tree performance. *Int. J. Adv. Res. Comput. Sci.* **2017**, *8*, 1856–1857. [CrossRef]
7. Tripathy, S.; Satpathy, M. SSD internal cache management policies: A survey. *J. Syst. Archit.* **2022**, *122*, 102334. [CrossRef]
8. Tan, L.; Wang, Y.; Yi, J.; Yang, F. Single-Instruction-Multiple-Data Instruction-Set-Based Heat Ranking Optimization for Massive Network Flow. *Electronics* **2023**, *12*, 5026. [CrossRef]
9. Tran, B.; Schaffner, B.; Myre, J.; Sawin, J.; Chiu, D. Exploring Means to Enhance the Efficiency of GPU Bitmap Index Query Processing. *Data Sci. Eng.* **2020**, *6*, 209–228. [CrossRef]
10. Kouassi, E.K.; Amagasa, T.; Kitagawa, H. Efficient Probabilistic Latent Semantic Indexing using Graphics Processing Unit. *Procedia Comput. Sci.* **2011**, *4*, 382–391. [CrossRef]
11. Gowanlock, M.; Rude, C.; Blair, D.M.; Li, J.D.; Pankratius, V. A Hybrid Approach for Optimizing Parallel Clustering Throughput using the GPU. *IEEE Trans. Parallel Distrib. Syst.* **2019**, *30*, 766–777. [CrossRef]
12. Anneser, C.; Kipf, A.; Zhang, H.; Neumann, T.; Kemper, A. Adaptive Hybrid Indexes. In Proceedings of the 2022 International Conference on Management of Data, Philadelphia, PA, USA, 12–17 June 2022. [CrossRef]
13. Sun, Y.; Zhao, T.; Yoon, S.; Lee, Y. A Hybrid Approach Combining R*-Tree and k-d Trees to Improve Linked Open Data Query Performance. *Appl. Sci.* **2021**, *11*, 2405. [CrossRef]
14. Kraska, T. Towards instance-optimized data systems. *Proc. VLDB Endow.* **2021**, *14*, 3222–3232. [CrossRef]
15. Sadri, Z.; Gruenwald, L.; Leal, E. Online Index Selection Using Deep Reinforcement Learning for a Cluster Database. In Proceedings of the 2020 IEEE 36th International Conference on Data Engineering Workshops (ICDEW), Dallas, TX, USA, 20–24 April 2020, pp. 158–161. [CrossRef]
16. Tan, Y.K.; Xu, X.; Liu, Y. Improved Recurrent Neural Networks for Session-based Recommendations. In Proceedings of the 1st Workshop on Deep Learning for Recommender Systems, Boston, MA, USA, 15 September 2016. [CrossRef]
17. Marcus, R.; Kipf, A.; van Renen, A.; Stoian, M.; Misra, S.; Kemper, A.; Neumann, T.; Kraska, T. Benchmarking learned indexes. *Proc. VLDB Endow.* **2020**, *14*, 1–13. [CrossRef]
18. Schab, S. The comparative performance analysis of selected relational database systems. *J. Comput. Sci. Inst.* **2023**, *28*, 296–303. [CrossRef]
19. Nambiar, R.; Wakou, N.; Carman, F.; Majdalany, M. Transaction processing performance council (TPC): State of the council 2010. In Proceedings of the Performance Evaluation, Measurement and Characterization of Complex Systems: Second TPC Technology Conference, TPCTC 2010, Singapore, 13–17 September 2010; Revised Selected Papers 2; Springer: New York, NY, USA, 2011; pp. 1–9.