Data Energy Saving via Approximate Nearest Neighbor Search in Hamming Distance

Xinyu Liang, Chunzhen Hu, Yuxuan Liu

1 Introduction

Non-volatile memory (NVM) is widely favored for its persistence, high density, and fast access speeds. However, like all other technologies, it has certain inherent limitations, one of which is the high cost of write operations. Compared to DRAM, each write operation in NVM consumes more energy and has lower write endurance, meaning that after a certain number of writes, the segment of the data medium may become unreliable.

To address this issue, the Hamming Tree method has emerged as a promising solution. By organizing memory locations based on their Hamming distance, the Hamming Tree allows for more judicious selection of write locations, thereby reducing the number of bit flips. This approach not only enhances write endurance but also improves energy efficiency, as fewer bit flips result in lower power consumption during write operations. The Hamming Tree effectively leverages the structural properties of data to optimize memory usage and extend the lifespan of NVM.

Although the Hamming Tree method effectively reduces bit flips, storing the Hamming Tree data structure requires significant space, making it impractical for large-scale NVM systems. To overcome this limitation, we propose integrating an approximate nearest neighbor search (NNS) tailored for NVM applications to balance efficiency and storage overhead.

Unlike traditional NNS methods that prioritize high accuracy and complex computations, our approach focuses on speed and simplicity. We employ Product Quantization (PQ), which splits large vectors into smaller sub-vectors and represents them with compressed IDs. Instead of the typical Euclidean distance, we use the Hamming distance to measure similarity, allowing us to quickly identify memory locations that closely match the data to be written.

By combining ideas from the Hamming Tree method with Product Quantization, our approach reduces storage requirements while efficiently selecting write locations. We evaluate this method using key metrics such as bit flip reduction, execution time, and Hamming distance percentage. The results show a 30% reduction in bit flips and a consistently low Hamming distance percentage, making it a practical solution for optimizing NVM performance.

2 Background

2.1 Motivation

Non-volatile memory (NVM) is a memory technology that retains data even when power is lost, bridging the gap between traditional DRAM (volatile memory) and storage devices like HDDs. NVM is widely used in various applications, such as data centers, consumer storage devices (e.g., SSDs and USB flash drives), enterprise storage, cloud computing, and the Internet of Things (IoT). Major semiconductor companies, including Intel, Micron, Samsung, Western Digital, and Toshiba, are actively developing and utilizing NVM technology.

NVM is favored for its persistence, high density, and fast access speed. However, it faces two significant challenges: low energy efficiency and low write endurance. Writing to NVM is energy-intensive, as flipping a single bit costs approximately 50 pJ, whereas writing an entire DRAM page requires only 1 pJ. Additionally, NVM devices have limited write endurance,

typically handling around 10^8 **to** 10^9 writes before becoming unreliable, compared to DRAM's write endurance of approximately 10^{15} writes.

These challenges highlight the need to reduce the number of bit flips during write operations to improve energy efficiency and prolong the lifespan of NVM devices.

2.2 Non Memory-Aware Solutions

To address the challenges of energy consumption and write endurance in NVM, two mainstream solutions have been developed. The first solution is Read-Before-Write (RBW), which ensures that only bits that differ between the new data and the existing data are written. This technique reduces unnecessary bit flips by reading the current data before performing the write operation and only flipping the bits that need to change.

The second solution is Minimizing Write Amplification, which aims to reduce the number of actual writes performed on the memory. Write amplification occurs when a small amount of data modification results in a disproportionately large number of write operations, particularly in systems that use techniques like logging or garbage collection. By optimizing data placement and reducing the frequency of block-level writes, this approach decreases the overall number of write operations and mitigates write endurance issues.

However, both of these solutions are non-memory-aware, meaning they reduce bit flips indirectly without leveraging detailed knowledge of the memory's internal state or content. Because these methods are not aware of the exact bit-level patterns within the memory, they can miss opportunities for further reducing bit flips. A memory-aware approach, such as the Hamming Tree method that we will discuss next, considers the content and structure of the memory to directly minimize bit flips during write operations, offering a more efficient solution for extending the lifespan and improving the energy efficiency of NVM devices.

2.3 Hamming Tree

2.3.1 Intuition

The core intuition behind the Hamming Tree method is to minimize bit flips when writing data to NVM. When selecting a write location, we aim to find a memory location that closely matches the data to be written, thereby reducing the number of bit flips. A straightforward way to quantify the difference between two values is through the Hamming distance. The Hamming distance is computed by performing a bitwise XOR between two values and counting the number of ones in the result, which represents the number of differing bits.

The Hamming Tree method is built on this idea and employs a comparator to compare the density of ones in two values. If one value has a higher density of ones on the right-hand side, it is considered larger than the other value. This comparator forms the basis for constructing a Hamming Tree, which is a variation of a B-tree of order three.

2.3.2 Implementations

The Hamming Tree resides in the DRAM and is not persistent; however, it can be reconstructed easily upon system startup. This is facilitated by flags set for each memory segment in the main table, allowing quick identification of available or occupied memory regions.

The Hamming Tree supports three main operations: PUT, DELETE, and UPDATE. The UPDATE operation is essentially a combination of a DELETE followed by a PUT. Therefore, in practice, there are only two distinct operations:

- **PUT:** Allocates space in the NVM and removes the address from the Hamming Tree.

- **DELETE:** Frees space in the NVM and adds the address back to the Hamming Tree.

If you are familiar with how memory allocation and deallocation (free) work in C, the PUT and DELETE operations in the Hamming Tree will seem similar to managing a free list.

Additionally, as you traverse the tree from the root to the leaves, the similarity between the data to be written and the available memory locations increases. While you may not always find an exact match, the deeper you go into the tree, the closer the match you will achieve. This allows for selecting an optimal write location even when a perfect match is unavailable.

For more details about the Hamming Tree method, refer to the original paper[2].

2.3.3 Limitations

Although the Hamming Tree method has reduced the bit flips significantly, it comes with a drawback: it requires a large amount of space to store the Hamming Tree data structure. This increased storage overhead can be a limiting factor, especially when dealing with large-scale systems or memory-constrained environments. The need for maintaining extensive tree structures results in higher memory consumption, which can offset the benefits gained from reducing bit flips.

Let's say we are working on a 1 TB NVM on an x64 machine, and each memory segment is 4 KB. That means there are:

$$\frac{1\,\text{TB}}{4\,\text{KB}} = 2^{28} \approx 268{,}435{,}456 \text{ segments.}$$

These segments are stored in a B-tree of order 3, where each node can have up to 5 keys, implying approximately 53,687,092 leaf nodes. Considering a branching factor of up to 6, the tree height would be around 10.

With internal and leaf nodes combined, the total number of nodes is on the order of tens of millions — roughly 64 million. Assuming each node requires about 100–120 bytes for keys, pointers, and overhead, the storage required for the data structure itself is:

$$64{,}000{,}000 \text{ nodes} \times 100\text{–}120 \text{ bytes per node} = 6\text{–}8 \text{ GB}.$$

Thus we can see that the Hamming Tree method is not suitable for large NVM. The significant storage overhead required to maintain the tree structure becomes impractical when dealing with terabyte-scale memory. As mentioned in the paper[2], when working with large NVM, it is better to map only a portion of the NVM. This approach serves as a compromise, reducing the storage requirements of the Hamming Tree method while still achieving reasonable performance gains in terms of reducing bit flips.

3 Our Method: Product Quantization

In the previous section, we discussed the Hamming Tree method and its limitation of requiring substantial memory despite reducing bit flips. To address this, we propose Product Quantization, a method that not only significantly reduces bit flips but also utilizes less space. By leveraging Product Quantization, we aim to achieve a more efficient and scalable solution for our project.

3.1 What is Product Quantization

Product Quantization is a technique used to efficiently compress and search high-dimensional vectors. The basic idea involves clustering the vector space into multiple subspaces and representing each subspace with a set of centroids. Each high-dimensional vector can then be

approximated by the centroid closest to it. This reduces storage requirements since the vector can be stored as a centroid index rather than its full representation.

The Product Quantization algorithm works as follows: First, we perform pre-training on all vectors in the non-volatile memory (NVM) using K-means clustering to determine the centroids for all subspaces. These centroids are stored in a codebook. When a new write query arrives, we split the vector and find the centroid in the codebook with the smallest Hamming distance to each subvector. The original vector is then represented by the centroid's ID. Finally, we combine all these IDs to form a much smaller vector, significantly reducing the storage size while preserving essential information.[1]

Product Quantization is widely used in areas like large-scale image retrieval, vector search, and deep learning model compression. Companies like Meta, Google, and Microsoft utilize Product Quantization in their large-scale search and indexing systems to improve performance and reduce storage costs.

In this project, we are utilizing Product Quantization because we are working with large vectors and require an efficient method for storage and search. Since we are not highly concerned about perfect matching rates, Product Quantization provides a balanced solution that efficiently reduces storage requirements while maintaining reasonable retrieval accuracy.

3.2 Comparison: HT and PQ

Let's run a real-world example to further explain the algorithm and compare it with the Hamming tree method. Suppose we're still working on a 1 TB NVM, where each vector size is 4 KB, matching a typical page size. By selecting a sub-vector size of 256 B, we create 16 subspaces (4096 bytes / 256 bytes) per vector. Each subspace contains 256 centroids, meaning each ID requires 8 bits (1 byte) for representation. The codebook storage, therefore, amounts to:

$$16 \text{ subspaces} \times 256 \text{ centroids} \times 256 \text{ bytes} = 1 \text{ MB}$$

For the encoded pages, each page is represented by 16 indices or IDs (16 bytes). With 2.44×10^8 pages in total, the overall storage required for the encoded pages sums up to:

$$2.44 \times 10^8 \text{ pages} \times 16 \text{ bytes} = 3.91 \text{ GB}$$

for the encoded pages sums up to 3.91 GB.

Compared to the Hamming Tree method, which also utilizes a 1TB NVM and 4KB memory segments, our approach requires only 3.91GB + 1MB to store the auxiliary data structures. This is a significant improvement, reducing storage requirements to about a half of the previous method. Later, we will test the bit flips reduction and prediction success rate in the experiments section to evaluate its effectiveness.

3.3 Our Design

In our design, we chose to use 256 centroids in each subspace with a sub-vector size of 16. We also work with 4KB vectors, which match the page size. By dividing 4096 by 16, we determine the number of sub-vectors to be 256 in our algorithm. During the pre-processing phase, we utilize Hamming distance to perform K-means clustering training and encode each page. A typical K-means clustering method uses Euclidean distance but we use Hamming distance here instead. The encoding process involves replacing each sub-vector with the ID of its nearest centroid. This allows us to effectively encode the pages on the fly. When accepting write queries, we first encode the incoming big vector and then identify the best matching page among all encoded pages to write the data.

4 Experiments

For our experiments, due to the lack of hardware support (physical NVM like Intel Optane Memory), we will simulate persistent memory by designating a portion of the DRAM to act as persistent storage. This approach will allow us to emulate the behavior of persistent memory within our existing hardware constraints and continue our research and development using this simulated environment.

4.1 Environment Setup

In this report, we used an Intel Core i7 processor running at 3.8 GHz with 8 cores, each equipped with an 8 MB L2 cache and a 16.5 MB L3 cache, and 121 GB of DRAM. To simulate persistent memory, we modified the grub file to map a 4 GB section of DRAM as persistent memory and rebooted the host. After rebooting, we mounted an EXT4 filesystem to the newly created device representing the persistent memory region. This device corresponds to the memmap regions specified in the grub configuration. More details can be found on the PMDK website[3].

4.2 Workflow

In our experiments, we first create four datasets with different distributions: uniform, latest, and hotspot. Each dataset is stored as a CSV file, where each line represents a key-value pair to simulate database behavior. Each query set contains 100K queries for both the default approach and our Product Quantization (PQ) algorithm.

Before running the queries, we perform a warm-up by filling the persistent memory with random data. We then compare the default behavior with our PQ algorithm. In the default approach, incoming queries from the CSV files are written to random locations. In contrast, our PQ algorithm selects the optimal page for each write based on the query's content.

For both approaches, we measure the total number of bit flips and the execution time. In PQ, we distinguish between the training phase and the execution phase, measuring the time for each separately. This distinction is important because, in real-world scenarios, the NVM is typically trained once, while the execution phase, which is processing incoming queries, can occur repeatedly over a long period. By measuring these phases separately, we gain a clearer understanding of the breakdown of time costs and can better assess the efficiency of the PQ approach during actual deployment. Additionally, for PQ, we measure the percentage of Hamming distance, which is the fraction of bit flips in all bits, to evaluate its effectiveness.

4.3 Results

We evaluated both the default and Product Quantization (PQ) methods across four datasets (Uniform, Zipfian, Latest, and Hotspot). The results show:

- **Default Method:** Consistently had higher bit flips and shorter execution times ($\sim$1.06s) but did not optimize write efficiency.
- **PQ Method:** Reduced the number of bit flips significantly but required separate training and execution times. Training took $\sim$397s, while execution was $\sim$273s. The percentage of Hamming distance was around 1% across all distributions.

Dist.	Method	Bit Flips	Time (s)	Hamming
Uniform	Default	48,243,860	1.067	N/A
	PQ	34,911,852	397.418(train), 273.929(exec)	1.00564%
Zipfian	Default	48,254,613	1.063	N/A
	PQ	32,932,322	396.093(train), 273.412(exec)	0.98871%
Latest	Default	48,320,243	1.064	N/A
	PQ	33,413,925	397.098(train), 273.696(exec)	1.02336%
Hotspot	Default	48,258,762	1.063	N/A
	PQ	32,841,947	399.923(train), 273.630(exec)	1.02092%

Table 1: Test Results for Default and PQ Methods

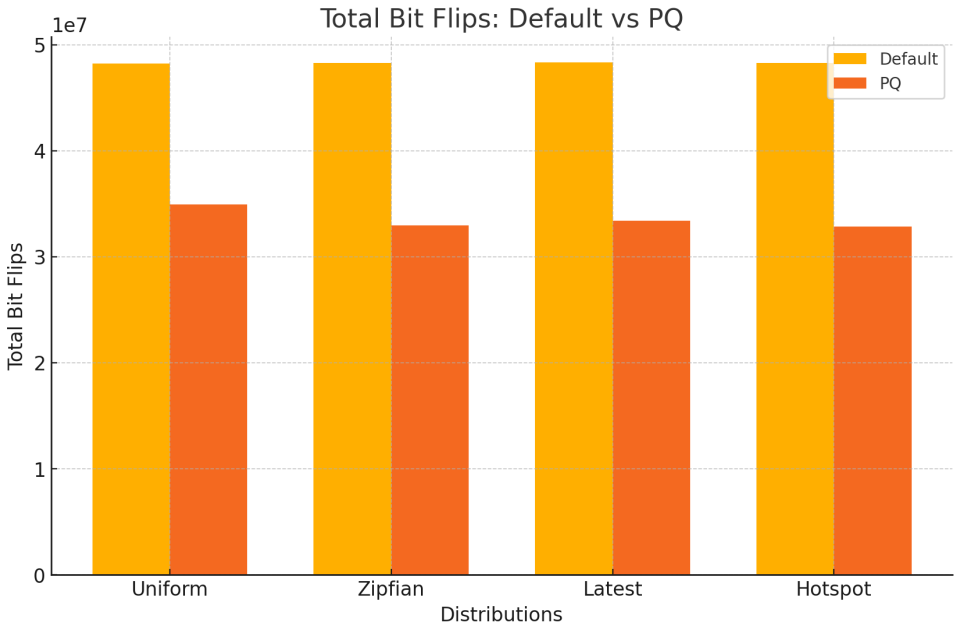

Figure 1: Total Bit Flips Comparison: Default vs PQ

Key Statistics

Our test results reveal that the PQ method achieves a 30.5% average reduction in total bit flips compared to the default method. This represents a good enough and decently effective improvement in write efficiency.

Interestingly, we observed that the reductions were consistent across all distributions (Uniform, Zipfian, Latest, and Hotspot), with no drastic differences between them. Additionally, the time breakdown for PQ remained consistent, with training taking about 397 seconds and execution around 273 seconds for all datasets.

Finally, the percentage of Hamming distance consistently hovered around 1%. This result is surprisingly low, indicating a much better prediction success rate than we initially expected for identifying the optimal write location.

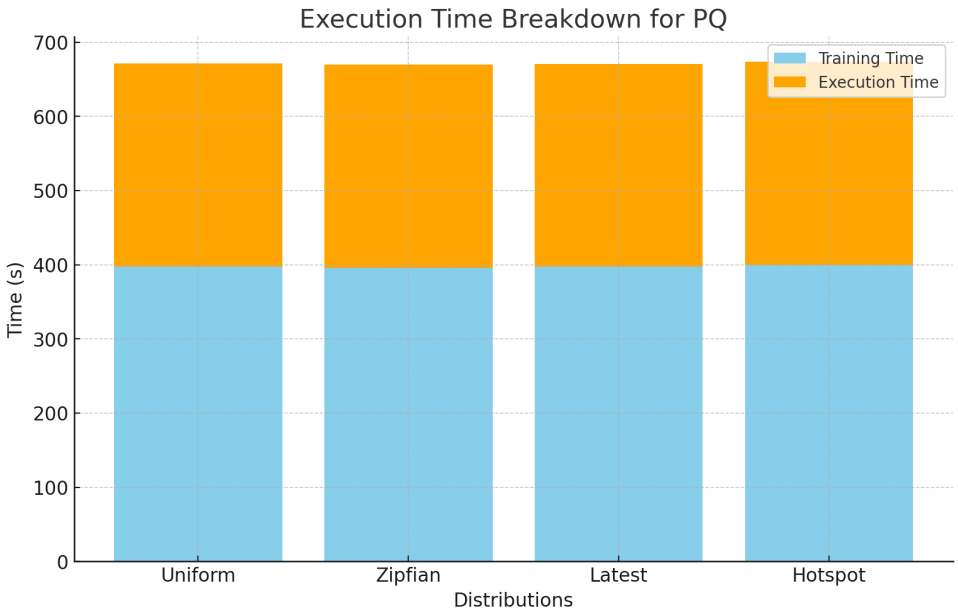

Figure 2: Execution Time Breakdown for PQ

5 Limitations

While our experiments demonstrate good performance and promising results, there are several limitations to consider:

1. **Hardware Limitations:** Our experiments were conducted in a simulated environment rather than on physical NVM hardware. This simulation may introduce discrepancies, as real NVM devices could exhibit different performance characteristics. Additionally, our simulation uses a relatively small 4 GB NVM, and scaling to larger capacities may yield different results.

2. **Dataset Limitations:** Initially, we intended to use the YCSB benchmark, which offers real-world datasets with various distributions. However, implementing this required a deeper understanding of database systems, such as intercepting and modifying queries at runtime. Due to these complexities, we generated our own datasets instead. Consequently, our results may differ from those obtained using the YCSB benchmark.

3. **Energy Measurement:** We did not measure energy consumption directly in joules. Although we planned to use tools like `perf` to measure energy costs, we faced challenges in implementation. Instead, we used execution time as a proxy for energy consumption. Future work could explore direct energy measurements to provide more accurate insights.

4. **Data Structure Persistence:** The Hamming Tree method, referenced in prior work, uses an external, augmented data structure. In contrast, our approach builds the data structure dynamically during runtime, and it is not persistent. In the future, integrating a persistent data structure within the database could improve performance and reliability.

5. **K-Means Algorithm:** Our current implementation uses a basic K-means clustering algorithm for training the NVM. More advanced clustering libraries or community-optimized algorithms could potentially enhance performance and accuracy. Exploring these alternatives in future work may lead to better results.

6. **Model Adaptation Over Time:** As more queries are processed, the NVM's state may evolve significantly from its initial condition. Retraining the model periodically could improve prediction accuracy. However, in our current implementation, we did not perform retraining, which may impact long-term performance.

Addressing these limitations in future research could help refine our approach and provide more robust and scalable results.

6 Conclusion

In this report, we explored the potential of reducing bit flips in non-volatile memory (NVM) to address its key challenges of energy efficiency and write endurance. We began by discussing the background of NVM, highlighting its advantages such as persistence, high density, and fast access speed, as well as its disadvantages, particularly high write energy costs and limited write endurance. This established the need for efficient bit flip reduction techniques.

We compared non-memory-aware approaches, such as Read-Before-Write (RBW) and minimizing write amplification, with memory-aware techniques. We introduced the Hamming Tree method, which leverages bit-level similarity to reduce bit flips but comes with certain drawbacks. These limitations led us to propose an alternative solution: Product Quantization (PQ). We explained the PQ technique, which works by splitting large vectors into smaller sub-vectors and representing them with compressed IDs, thereby optimizing write operations.

Our experimental evaluation demonstrated the effectiveness of the PQ approach. We achieved a 30% reduction in bit flips, and the measured Hamming distance between the write queries and the selected optimal pages was consistently around 1%. This surprisingly low Hamming distance indicates that the chosen write locations closely match the data to be written, effectively minimizing bit flips. While the results are promising, we also discussed the limitations of our approach, such as hardware constraints, dataset choices, and potential areas for future improvements.

Overall, our work highlights the potential of memory-aware techniques like Product Quantization in enhancing the energy efficiency and lifespan of NVM devices. Future research could further refine these methods and explore their application on real NVM hardware for broader deployment.

References

[1] H. Jégou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(1):117–128, 2011.

[2] S. Kargar and F. Nawab. Hamming tree: The case for energy-aware indexing for nvms. *Proc. ACM Manag. Data*, 1(2), June 2023.

[3] Persistent Memory Development Kit (PMDK). Creating development environments: Linux environments - linux memmap, 2023. Accessed: 2023-12-10.