# Cache locality is not enough: High-Performance Nearest Neighbor Search with Product Quantization Fast Scan

Fabien André, Anne-Marie Kermarrec, Nicolas Le Scouarnec

# Cache locality is not enough: High-Performance Nearest Neighbor Search with Product Quantization Fast Scan

Fabien André
Technicolor
fabien.andre@technicolor.com

Anne-Marie Kermarrec
Inria
anne-marie.kermarrec@inria.fr

Nicolas Le Scouarnec
Technicolor
nicolas.lescouarnec@technicolor.com

## ABSTRACT

Nearest Neighbor (NN) search in high dimension is an important feature in many applications (e.g., image retrieval, multimedia databases). Product Quantization (PQ) is a widely used solution which offers high performance, i.e., low response time while preserving a high accuracy. PQ represents high-dimensional vectors (e.g., image descriptors) by compact codes. Hence, very large databases can be stored in memory, allowing NN queries without resorting to slow I/O operations. PQ computes distances to neighbors using cache-resident lookup tables, thus its performance remains limited by (i) the many cache accesses that the algorithm requires, and (ii) its inability to leverage SIMD instructions available on modern CPUs.

In this paper, we advocate that cache locality is not sufficient for efficiency. To address these limitations, we design a novel algorithm, PQ Fast Scan, that transforms the cache-resident lookup tables into small tables, sized to fit SIMD registers. This transformation allows (i) in-register lookups in place of cache accesses and (ii) an efficient SIMD implementation. PQ Fast Scan has the exact same accuracy as PQ, while having 4 to 6 times lower response time (e.g., for 25 million vectors, scan time is reduced from 74ms to 13ms).

## 1. INTRODUCTION

Nearest Neighbor (NN) search in high-dimensional spaces is an important feature in many applications including machine learning, multimedia databases and information retrieval. Multimedia data, such as audio, images or videos can be represented as feature vectors, characterizing their content. Finding a multimedia object similar to a given query object therefore involves representing the query object as a high-dimensional vector and finding its nearest neighbor in the feature vector space. While efficient methods exist to solve the NN search problem in low-dimensional spaces, the notorious *curse of dimensionality* challenges these solutions in high dimensionality. As the dimensionality increases, these methods are outperformed by bruteforce linear scans and large databases make this problem even more salient. To tackle this issue, the research community has focused on Approximate Nearest Neighbor (ANN) search, which aims at finding close enough vectors instead of the exact closest ones.

Locality Sensitive Hashing (LSH) [11, 8] has been proposed to solve the ANN problem. Yet, its significant storage overhead and important I/O cost limit its applicability to large databases. Recent advances in LSH-based approaches [19, 26] deal with these two issues but ANN search with these solutions still requires I/O operations. We focus on an alternative approach, named Product Quantization (PQ) [14, 27]. PQ is unique in that it stores database vectors as compact codes, allowing very large databases to be stored entirely in memory. In most cases, vectors can be represented by 8-byte codes, enabling a single commodity server equipped with 256 GB RAM to store 32 billion vectors in memory. Therefore, ANN search with PQ requires *no* I/O operations. The key idea behind PQ is to divide each vector into $m$ distinct sub-vectors, and encode each sub-vector separately. To answer ANN queries, PQ computes the distance between the query vector and a large number of database vectors using cache-resident *lookup tables*. This process, known as *PQ Scan*, has a high CPU cost and is the focus of this paper.

As it relies on main memory, PQ Scan is able to scan hundreds of millions of database vectors per second. However, because it performs many table lookups, PQ Scan is not able to fully leverage the capabilities of modern CPUs. Its CPU cost therefore remains high, as reported in [19]. In particular, PQ Scan cannot leverage SIMD instructions, which are yet crucial for performance. In this paper, through the example of PQ Scan, we investigate why algorithms based on lookup tables cannot leverage SIMD instructions. We propose alternatives to cache-resident lookup tables, enabling better performance. Based on our findings, we design a novel algorithm, named PQ Fast Scan. PQ Fast Scan achieves 4-6× better performance than PQ Scan while returning the exact same results. More specifically, this paper makes the following contributions:

- We extensively analyze PQ Scan performance. Our study shows that sizing lookup tables to fit the L1 cache is not enough for performance. We also demonstrate that PQ Scan cannot be implemented efficiently with SIMD instructions, even using gather instructions available in the latest generations of Intel CPUs.
- We design PQ Fast Scan that addresses these issues. The key idea behind PQ Fast Scan is to build small


*Proceedings of the VLDB Endowment,* Vol. 9, No. 4

tables, sized to fit SIMD registers, that can be looked up using fast SIMD instructions. We use these small tables to compute lower bounds on distances and avoid unnecessary accesses to the cache-resident lookup tables. This technique allows pruning over 95% of L1 cache accesses and thus provides a significant performance boost. To build these small tables, we rely on: (i) grouping of similar vectors, (ii) computation of minimum tables and (iii) quantization of floating-point distances to 8-bit integers.

- We implement PQ Fast Scan on Intel CPUs and evaluate its performance on the public ANN_SIFT1B dataset of high-dimensional vectors. We analyze parameters that impact its performance, and experimentally show that it achieves a 4-6× speedup over PQ Scan.
- We discuss the application of the techniques used in PQ Fast Scan beyond the context of ANN search.

# 2. BACKGROUND

This section describes (i) how Product Quantization (PQ) is used to represent high-dimensional vectors by compact codes and (ii) the various steps involved in answering an ANN query with PQ.

## 2.1 Product Quantization

Product Quantization builds on vector quantizers to represent high-dimensional vectors by compact codes [14]. A vector quantizer is a function $q$ that maps a $d$-dimensional vector $x$ to a $d$-dimensional vector $c_i$ belonging to a predefined set of vectors $\mathcal{C}$. Vectors $c_i$ are called *centroids*, and the set of centroids $\mathcal{C}$, of size $k$, is the *codebook*.

$$q : \mathbb{R}^d \rightarrow \mathcal{C} = (c_0, \ldots, c_{k-1})$$
$$q(x) = \mathcal{C}[i] = c_i$$

We consider Lloyd-optimal quantizers which map vectors to their closest centroids and can be built using k-means [20].

$$\mathrm{q}(x) = \arg\min_{c_i \in \mathcal{C}} ||x - c_i||$$

Vector quantizers allow representing vectors by compact codes by using the index $i$ of their closest centroid $c_i$ as representative. This enables a 128-dimensional vector of floats ($128 \times 32$ bits or 512 bytes of memory) to be represented by the index of its closest centroid (a single 64-bit integer or 8 bytes of memory).

$$\mathrm{code}(x) = i, \text{ such that } q(x) = \mathcal{C}[i]$$

To maintain the quantization error low, it is necessary to build a quantizer with a high number of centroids (e.g., $k = 2^{64}$ centroids). Yet, building such a quantizer is intractable both in terms of processing and memory requirements.

Product Quantization (PQ) addresses this issue by dividing an input vector $x$ of dimensionality $d$ into $m$ distinct sub-vectors $u_j(x), 0 \leq j < m$, and quantizing each sub-vector separately using distinct lower complexity sub-quantizers. Each sub-vector $u_j(x)$ has a dimensionality $d^* = d/m$, where $d$ is a multiple of $m$.

$$x = (\underbrace{x_0, \ldots, x_{d^*-1}}_{u_0(x)}, \ldots, \underbrace{x_{d-d^*}, \ldots, x_{d-1}}_{u_{m-1}(x)})$$

A product quantizer $q_p$ uses $m$ sub-quantizers to quantize the input vector $x$. Each sub-quantizer $q_j, 0 \leq j < m$ is a vector quantizer of dimensionality $d^*$, with a distinct codebook $\mathcal{C}_j$.

$$q_j : \mathbb{R}^{d^*} \rightarrow \mathcal{C}_j = (c_{j,0}, \ldots, c_{j,k^*-1})$$

A product quantizer $q_p$ maps an input vector $x$ of dimensionality $d$ as follows:

$$\begin{aligned} q_p : \mathbb{R}^d &\rightarrow \mathcal{C}_0 \times \cdots \times \mathcal{C}_m \\ q_p(x) &= (q_0(u_0(x)), \ldots, q_{m-1}(u_{m-1}(x)) \\ &= (\mathcal{C}_0[i_0], \ldots, \mathcal{C}_m[i_m]) \end{aligned}$$

The product quantizer $q_p$ can be used to represent an input vector $x$ by a compact code, pqcode($x$), by concatenating the indexes of the $m$ centroids returned by the $m$ sub-quantizers $q_j$:

$$\mathrm{pqcode}(x) = (i_0, \ldots, i_m),$$
$$\text{such that } q_p(x) = (\mathcal{C}_0[i_0], \ldots, \mathcal{C}_m[i_m])$$

We only consider product quantizers where sub-quantizers codebooks $\mathcal{C}_j$ have the same size $k^*$. The main advantage of product quantizers is that they are able to produce a large number of centroids $k$ from sub-quantizers with a lower number of centroids $k^*$, such that $k = (k^*)^m$. For instance, a product quantizer with 8 sub-quantizers of $2^8$ centroids has a total number of $k = (2^8)^8 = 2^{64}$ centroids. We focus on product quantizers with $2^{64}$ centroids as they offer a good tradeoff between complexity and quality for NN search [14].

We introduce the notation PQ $m\times \log_2(k^*)$ to designate a product quantizer with $m$ sub-quantizers and $k^*$ centroids per sub-quantizer. Thus, PQ 8×8 designates a product quantizer with 8 sub-quantizers and $2^8 = 256$ centroids per sub-quantizer. Any $(m, k^*)$ configuration such that $m\times\log_2(k^*) = 64$ allows building a product quantizer with $2^{64}$ centroids. The $m$ and $k^*$ parameters impact (i) the complexity of learning the product quantizer, (ii) its accuracy and (iii) the memory representation of database vectors. Lower values of $m$, and thus higher values of $k^*$ provide product quantizers with a lower quantization error at the price of an increased learning complexity. Database vectors are stored as pqcodes, which consist of $m$ indexes of $\log_2(k^*)$ bits. Therefore, in the remainder of this paper, we refer to pqcodes stored in the database as database vectors, or simply vectors. Figure 1 shows the memory representation of 6 database vectors, $p, \cdots, u$ encoded with a PQ 8×8 quantizer. Each vector is composed of 8 indexes of 8 bits.

| | 0 | | | | | | | 7 |
|---|---|---|---|---|---|---|---|---|
| $p$ | 01 | 03 | 02 | 05 | 06 | 09 | 04 | 08 |
| $q$ | 3f | 11 | 21 | 00 | 01 | f2 | 12 | 11 |
| $r$ | 04 | 0c | 0e | 1a | f1 | 0f | a9 | 17 |
| $s$ | f6 | ff | f6 | f0 | 23 | 0b | b6 | 2f |
| $t$ | 37 | 1a | 21 | 00 | 32 | 8b | e9 | 03 |
| $u$ | f5 | fc | ff | f1 | 46 | 33 | cf | 2c |
| | ⋮ | | | | | | | |

**Figure 1: Database vectors in memory**

## 2.2 ANN Search with Product Quantization

Nearest Neighbor Search requires computing distances between vectors stored in the database and a query vector $y$. We consider squared distances as they avoid a square root

**Algorithm 1** Nearest Neighbor Search with PQ

```
1: function NNS(y, database)
2:     part ← INDEX_GET_PARTITION(y, database)   ▷ Step 1
3:     D ← COMPUTE_DISTANCE_TABLES(y)            ▷ Step 2
4:     PQSCAN(D, part)                           ▷ Step 3
5: end function

6: function PQSCAN(D, part)
7:     min ← ∞
8:     pos ← 0
9:     for i ← 0 to |part| − 1 do
10:        p ← part_i                  ▷ i^th database vector
11:        d ← PQDISTANCE(p, D)
12:        if d < min then
13:            min ← d
14:            pos ← i
15:        end if
16:    end for
17:    return min, pos
18: end function

19: function PQDISTANCE(p, D)
20:    d ← 0
21:    for j ← 0 to m − 1 do
22:        index ← p[j]            ▷ index of the j^th centroid
23:        d ← d+D_j[index]
24:    end for
25:    return d
26: end function
```

computation while preserving the order. The Asymmetric Distance Computation (ADC) method allows computing the distance between a query vector $y$ and a database vector $p$, without quantizing the query vector [14]. ADC approximates the distance between the query vector $y$ and a database vector $p$ by:

$$\tilde{\mathrm{d}}(\mathrm{p},\mathrm{y}) = \sum_{j=0}^{m-1} \mathrm{d}(u_j(y), \mathcal{C}_j[p[j]]) \quad (1)$$

where $\mathrm{d}(u_j(y), \mathcal{C}_j[p[j]])$ is the squared distance between the $j^{\text{th}}$ sub-vector of the query vector $y$ and $\mathcal{C}_j[p[j]]$, the $j^{\text{th}}$ centroid associated with the database vector $p$.

To allow fast ANN queries, the IVFADC [14] system has been proposed. This system adds an inverted index (IVF) on top of the product quantizer and ADC. The index is built from a basic vector quantizer, named *coarse quantizer*. Each Voronoi cell of the coarse quantizer forms a *partition* of the database. Answering an ANN query with IVFADC involves three steps (Algorithm 1):

1. Selecting a partition of the database using the index. The selected partition corresponds to the Voronoi cell of the coarse quantizer where the query vector lies.
2. Computing distance tables, which are used to speed up ADC computations.
3. Scanning the partition, i.e., computing the ADC between the query vector and all vectors of the partition. We name this step PQ Scan.

PQ achieves high recall by scanning a large number of vectors. The selected partition typically comprises thousands

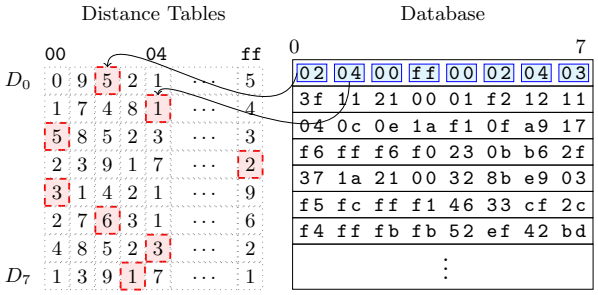



**Figure 2: PQ distance computation**

to millions of vectors, depending on the database size and the index parameters. We focus on very large databases and partitions exceeding 3 million vectors. In this case, Step 1 and 2 account for less than 1% of the CPU time while Step 3 uses more than 99% of the CPU time.

Once a partition has been selected, $m$ distance tables, $D_j$, $0 \leq j < m$, are computed (Step 2). These distance tables are specific to a given query vector. Each $D_j$ distance table is composed of the distance between the $j^{\text{th}}$ sub-vector of the query vector $y$ and every centroid of the $j^{\text{th}}$ sub-quantizer:

$$D_j = \left(d\left(u_j(y), \mathcal{C}_j[0]\right), \cdots, d\left(u_j(y), \mathcal{C}_j[k^*-1]\right)\right) \quad (2)$$

We omitted the definition of COMPUTE_DISTANCE_TABLES in Algorithm 1 for the sake of clarity but it would correspond to an implementation of Equation (2). Using these distance tables, the ADC equation (1) can be rewritten as:

$$\tilde{\mathrm{d}}(\mathrm{p},\mathrm{y}) = \sum_{j=0}^{m-1} D_j[p[j]] \quad (3)$$

PQ Scan (Step 3) iterates over all vectors (Algorithm 1, line 9) of the selected partition, and computes the ADC between the query vector and every vector of the partition using the PQDISTANCE function (Algorithm 1, line 11). The PQDISTANCE function is an implementation of Equation (3). Figure 2 shows the pqdistance computation between the query vector and the first database vector. The first centroid index ($p[0] = 02$) is used to look up a value in the first distance table ($D_0$), the second centroid index ($p[1] = 04$) is used to lookup a value in the second distance table ($D_1$) etc. All looked up values are then added to compute the final distance.

## 3. PQ SCAN LIMITATIONS

In this section, we show that despite its low apparent complexity, PQ Scan cannot be implemented efficiently on CPUs. We identify two fundamental bottlenecks that limit its performance: (i) the many cache accesses it performs and (ii) the impossibility to implement it efficiently using SIMD. Identification of these bottlenecks is key to designing our novel algorithm that overcomes PQ Scan limitations.

### 3.1 Memory Accesses

As PQ Scan parallelizes naturally over multiple queries by running each query on a different core, we focus on single-core performance. PQ Scan computes the pqdistance between the query vector and each database vector, which accounts for almost all CPU cycles consumed. The number

of operations required for each pqdistance computation depends on the $m$ parameter of the product quantizer. Each pqdistance computation involves:

- $m$ memory accesses to load centroid indexes $p[j]$ (Algorithm 1, line 22) [*mem1*]
- $m$ memory accesses to load $D_j$[index] values from distance tables (Algorithm 1, line 23) [*mem2*]
- $m$ additions (Algorithm 1, line 23)

We distinguish between two types of memory accesses: *mem1*, accesses to centroid indexes and *mem2*, accesses to distance tables as they may hit different cache levels. We analyze the locality of these two types of memory accesses. *Mem1* accesses always hits the L1 cache thanks to *hardware prefetchers* included in modern CPUs. Indeed, hardware prefetchers are able to detect sequential memory accesses and prefetch data to the L1 cache. We access p[j] values sequentially, i.e., we first access p[0] where p is the first database vector, then p[1] until p[m-1]. Next, we perform the same accesses on the second database vector, until we have iterated on all database vectors.

The cache level accessed by the *mem2* accesses depends on the $m$ and $k^*$ parameters of the product quantizer. Indeed, the size of distance tables, which impacts the cache level they can be stored in, is given by $m \times k^* \times$ sizeof(`float`). To obtain a product quantizer with $2^{64}$ centroids, which is important for accuracy, $m \times \log_2(k^*) = 64$. Therefore, smaller $m$ values lead to less memory accesses and additions but imply higher $k^*$ values and thus larger distance tables, which are stored in higher cache levels. Table 1 summarizes the properties of the different cache levels as well as the product quantizer configurations of which they can store the distance tables.

**Table 1: Cache level properties (Nehalem-Haswell)**

| | L1 | L2 | L3 |
|---|---|---|---|
| Latency (cycles) | 4-5 | 11-13 | 25-40 |
| Size | 32KiB | 256KiB | 2-3MiB × nb cores |
| PQ Configurations | PQ 16×4<br>PQ 8×8 | | PQ 4×16 |

PQ 16×4 is not interesting because it requires more memory accesses than PQ 8×8 and distance tables can be stored in the L1 cache for both configurations. PQ 4×16 requires two times less memory accesses than PQ 8×8, but PQ 4×16 distance tables are stored in the L3 cache which has a 5 times higher latency than the L1 cache. Overall, PQ 8×8 provides the best performance tradeoff, and is the most commonly used configuration in the literature [14, 27, 4, 10, 21]. Thus, from now on, we focus exclusively on PQ 8×8.

We use performance counters to analyze the performance of different PQ Scan implementations experimentally (Figure 3). For all implementations, the number of cycles with pending load operations (cycles w/ load) is almost equal to the number of cycles, which confirms that PQ Scan is a memory-intensive algorithm. We also measured the number of L1 cache misses (not shown on Figure 3). L1 cache misses represent less than 1% of memory accesses for all implementations, which confirms that both *mem1* and *mem2* accesses hit the L1 cache. The naive implementation of PQ Scan (Algorithm 1) performs 16 L1 loads per scanned vector: 8 *mem1* accesses and 8 *mem2* accesses. The authors of [14] distribute the libpq library[1], which includes an optimized implementation of PQ Scan. We obtained a copy of libpq under a commercial licence. Rather than loading 8 centroid indexes of 8 bits each (*mem1* accesses), the libpq implementation of PQ Scan loads a 64-bit word into a register, and performs 8-bit shifts to access individual centroid indexes. This allows reducing the number of *mem1* accesses from 8 to 1. Therefore, the libpq implementation of PQ Scan performs 9 L1 loads per scanned vector: 1 *mem1* access and 8 *mem2* accesses. However, overall, the libpq implementation is slightly slower than the naive implementation on our Haswell processor. Indeed, the increase in the number of instructions offsets the increase in IPC (Instructions Per Cycle) and the decrease in L1 loads.

## 3.2 Inability to Leverage SIMD Instructions

In addition to cache accesses, PQ Scan requires $m$ additions per pqdistance computation. We evaluate the applicability of SIMD instructions to reduce the number of instructions and CPU cycles devoted to additions. SIMD instructions perform the same operation, e.g., additions, on multiple data elements in one instruction. To do so, SIMD instructions operate on wide registers. SSE SIMD instructions operate on 128-bit registers, while more recently introduced AVX SIMD instructions operate on 256-bit registers [2]. SSE instructions can operate on 4 floating-point *ways* (4×32 bits, 128 bits) while AVX instructions can operate on 8 floating-point *ways* (8×32 bits, 256 bits). In this subsection, we consider AVX instructions as they provided the best results in our experiments. We show that PQ Scan structure prevents an efficient use of SIMD instructions.

To enable the use of fast *vertical* SIMD additions, we compute the pqdistance between the query vector and 8 database vectors at a time, designated by the letters $a$ to $h$. We still issue 8 addition instructions, but each instruction involves 8 different vectors, as shown on Figure 4. Overall, the number of instructions devoted to additions is divided by 8. However, the gain in cycles brought by the use of SIMD additions is offset by the need to set the ways of SIMD registers one by one. SIMD processing works best when all values in all ways are *contiguous* in memory and can be loaded in one instruction. Because they were looked up in a table, $D_0[a[0]], D_0[b[0]], \cdots, D_0[h[0]]$ values are not contiguous in memory. We therefore need to insert $D_0[a[0]]$ in the first way of the SIMD register, then $D_0[b[0]]$ in the second way etc. In addition to memory accesses, doing so requires many SIMD instructions, some of which have high latencies. Overall, this offsets the benefit provided by SIMD additions. This explains why algorithms relying on lookup tables, such as PQ Scan, hardly benefit from SIMD processing. Figure 3 shows that the AVX implementation of PQ Scan requires slightly less instructions than the naive implementation, and is only marginally faster.

To tackle this issue, Intel introduced a `gather` SIMD instruction in its latest architecture, Haswell [2]. Given an SIMD register containing 8 indexes and a table stored in memory, `gather` looks up the 8 corresponding elements from the table and stores them in a register, in just one instruction. This avoids having to use many SIMD instructions to set the 8 ways of SIMD registers. Figure 5 shows how `gather`

[1] `http://people.rennes.inria.fr/Herve.Jegou/projects/ann.html`

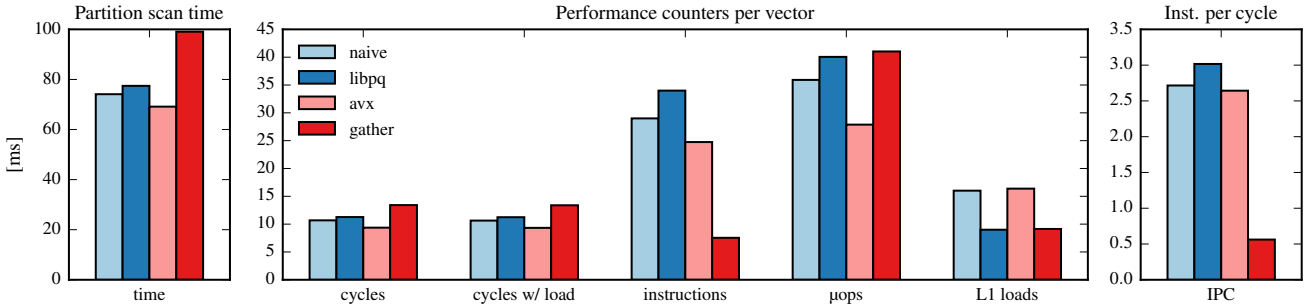



**Figure 3: Scan times and performance counters for 4 implementations of PQ Scan (25M vectors)**

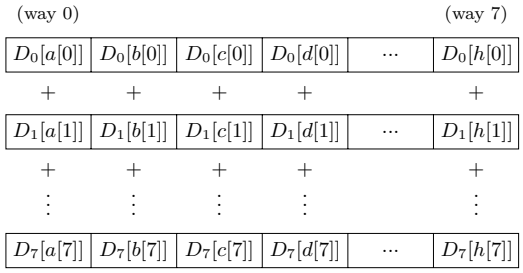



**Figure 4: PQ Scan with SIMD vertical adds**

can be used to look up 8 values in the first distance table ($D_0$). To efficiently use `gather`, we need $a[0], \cdots, h[0]$ to be stored contiguously in memory, so that they can be loaded in one instruction. To do so, we transpose the memory layout of the database presented in Figure 1. We store the first components of 8 vectors contiguously ($a[0], \cdots, h[0]$), followed by the second components of the same 8 vectors ($a[1], \cdots, h[1]$) etc., instead of storing all components of the first vector ($a[0], \cdots, a[7]$), followed by the components of the second vector ($b[0], \cdots, b[7]$). This also allows to reduce the number of *mem1* accesses from 8 to 1, similarly to the libpq implementation.

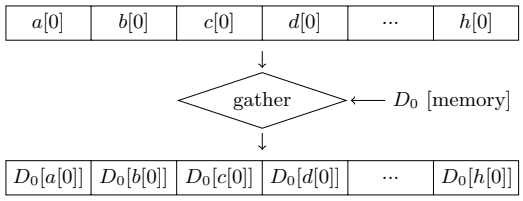



**Figure 5: SIMD gather operation**

However, the gather implementation of PQ Scan is slower than the naive version (Figure 3), which can be explained by several factors. First, even if it consists of only one instruction, `gather` performs 1 memory access for each element it loads, which implies suffering memory latencies. Second, at the hardware level, `gather` executes 34 µops[2] (Table 2) where most instructions execute only 1 µop. Figure 3 shows that the gather implementation has a low instructions count

[2]Micro-operations (µops) are the basic operations executed by the processor. Instructions are sequences of µops.

**Table 2: Instruction properties (Haswell)**

| Inst. | Lat. | Through. | µops | # elem | elem size |
|---|---|---|---|---|---|
| `gather` | 18 | 10 | 34 | no limit | 32 bits |
| `pshufb` | 1 | 0.5 | 1 | 16 | 8 bits |

but a high µops count. For other implementations, the number of µops is only slightly higher than the number of instructions. It also has a high latency of 18 cycles and a throughput of 10 cycles (Table 2), which means it is necessary to wait 10 cycles to pipeline a new `gather` instruction after one has been issued. This translates into poor pipeline utilization, as shown by the very low IPC of the gather implementation (Figure 3). In its documentation, Intel acknowledges that `gather` instructions may only bring performance benefits in specific cases [1] and other authors reported similar results [13].

# 4. PQ FAST SCAN

In this section, we present PQ Fast Scan, a novel algorithm we design that overcomes the limitations of PQ Scan. PQ Fast Scan performs less than 2 L1 cache accesses per scanned vector and allows additions to be implemented efficiently using SIMD instructions. By design, PQ Fast Scan returns exactly the same results as PQ Scan with a PQ $8\times8$ quantizer while performing 4-6 times faster.

## 4.1 Description

The key idea behind PQ Fast Scan is to use *small tables*, sized to fit SIMD registers instead of the cache-resident distance tables. These small tables are used to compute *lower bounds* on distances, without accessing the L1 cache. Therefore, lower bounds computations are fast. In addition, they are implemented using SIMD additions, further improving performance. We use lower-bound computations to prune slow pqdistance computations, which access the L1 cache and cannot benefit from SIMD additions. Figure 6 shows the processing steps applied to every database vector $p$. The $\otimes$ symbol means we discard the vector $p$ and move to the next database vector. The *min* value is the distance of the query vector to the current nearest neighbor. Our experimental results on SIFT data show that PQ Fast Scan is able to prune 95% of pqdistance computations.

To compute lower bounds, we need to look up values in small tables stored in SIMD registers. To do so, we use the `pshufb` instruction, which is key to PQ Fast Scan performance. Similarly to `gather`, `pshufb` looks up values in a

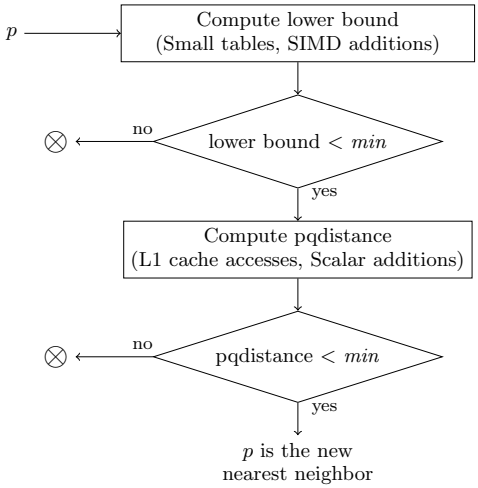



**Figure 6: Overview of PQ Fast Scan**

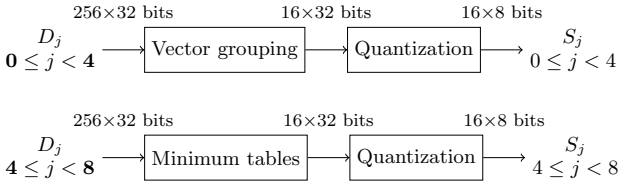



**Figure 7: Small tables building process**

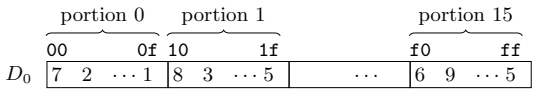



**Figure 8: Portions of the first distance table**

table corresponding to indexes stored in an SIMD register. However, in the case of `gather`, the table is stored in memory while in the case of `pshufb`, the table is stored in an SIMD register. This allows `pshufb` to have a much lower latency than `gather`, but limits the size of small tables to 16 elements of 8 bits each (16×8 bits, 128 bits). Furthermore, `pshufb` uses 16 indexes while `gather` uses only 8 indexes. Table 2 summarizes the properties of `gather` and `pshufb`.

To compute pqdistances, the original PQ Scan algorithm (with a PQ 8×8 quantizer) uses 8 distance tables $D_j$, $0 \leq j < 8$, and each distance table comprises 256 elements of 32 bits. Hence, one distance table (256×32 bits) does not fit into an SIMD register, which is why we need to build small tables. Just like there are 8 distance tables $D_j$, $0 \leq j < 8$, we build 8 small tables $S_j$, $0 \leq j < 8$. Each small table $S_j$ is stored in a distinct SIMD register and is built by applying transformations to the corresponding $D_j$ table.

To build 8 small tables suitable to compute lower bound on distances, we combine three techniques: (1) *vector grouping*, (2) computation of *minimum tables* and (3) *quantization of distances*. The first two techniques, vector grouping and computation of minimum tables, are used to build tables of 16 elements (16×32 bits). The third technique, quantization of distances, is used to shrink each element to 8 bits (16×32 bits → 16×8 bits). We group vectors and quantize distances to build the *first four* small tables, $S_0, \cdots, S_3$. We compute minimum tables and quantize distances to build the *last four* small tables, $S_4, \cdots, S_7$. Figure 7 summarizes this process.

## 4.2 Vector Grouping

Database vectors are pqcodes, which consist of 8 components of 8 bits (Figure 9a). When computing the pqdistance, each component is used as an index in the corresponding distance table, e.g., the 1st component is used as an index in the 1st distance table. The key idea behind *vector grouping* is to group vectors such that all vectors belonging to a *group* hit the same *portion* of 16 elements of a distance table.

We focus on the first distance table, $D_0$. We group vectors on their first component and divide the $D_0$ table into 16 portions, of 16 elements each (Figure 8). All database vectors $p$ having a first component p[0] between `00` and `0f` (0 to 15) will trigger lookups in portion 0 of $D_0$ when computing the pqdistance. These vectors form group 0. All vectors having a first component between `10` and `1f` (16 to 31) will trigger lookups in portion 1 of $D_0$. These vectors form the group 1. We define 16 groups in this way. Each group is identified by an integer $i$, and contains database vectors $p$ such that:

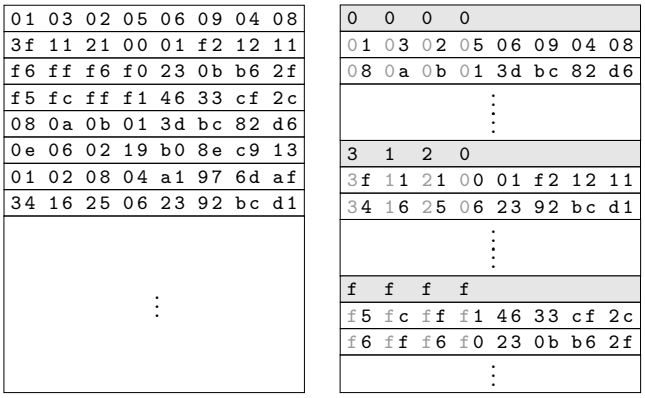
$$16(i-1) \leq p[0] < 16i$$

and only requires the portion $i$ of the first distance table, $D_0$. We apply the same grouping procedure on the 2nd, 3rd and 4th components. Eventually, each group is identified by four integers $(i_0, i_1, i_2, i_3)$, each belonging to $[0; 16[$ and contains vectors such that:

$$16(i_0-1) \leq p[0] < 16i_0 \ \wedge \ 16(i_1-1) \leq p[1] < 16i_1 \ \wedge$$
$$16(i_2-1) \leq p[2] < 16i_2 \ \wedge \ 16(i_3-1) \leq p[3] < 16i_3$$

Figure 9b shows a database where all vectors have been grouped. We can see that all vectors in the group $(3, 1, 2, 0)$ have a first component between `30` and `3f`, a second component between `10` and `1f`, etc. To compute the pqdistance of the query vector to any vector of a group $(i_0, i_1, i_2, i_3)$, we only need a portion of $D_0$, $D_1$, $D_2$ and $D_3$. Before scanning a group, we load the relevant portions of $D_0, \cdots, D_3$ into 4 SIMD registers to use them as the small tables $S_0, \cdots, S_3$. This process is shown by solid arrows on Figure 13. We do not apply grouping on all 8 components so as not to create too small groups. The average size $s$ of a group is given by $s = n/16^c$, where $n$ is the number of vectors in the scanned partition and $c$ the number of components used for grouping. For best performance, $s$ should exceed about 50 vectors. Indeed, before scanning a group, we load portions of distances tables into SIMD registers, which is costly. If the group comprises less than 50 vectors, a large part of the CPU time is spent loading distance tables. This is detrimental to performance, as shown by our experimental results (Section 5.6). The minimum partition size $n_{min}(c)$ to be able to group vectors on $c$ components is therefore given by $n_{min}(c) = 50 \cdot 16^c$, and increases *exponentially* with the number of components used for grouping. In this paper, we target partitions of $n = 3.2 - 25$ million vectors, therefore we always group on $c = 4$ components.

Grouping vectors also allows decreasing the amount of memory consumed by the database by approximatively 25%. In a group, the 1st component of all vectors has the same 4 most significant bits. As we apply grouping on the 4 first components, their 2nd, 3rd and 4th components also have




(a) Unordered vectors (b) Grouped vectors

**Figure 9: Vector grouping**

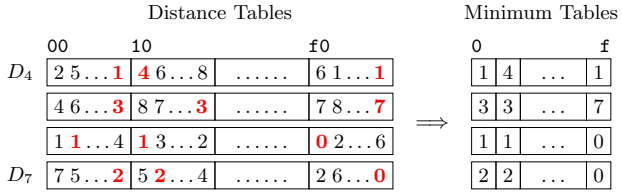


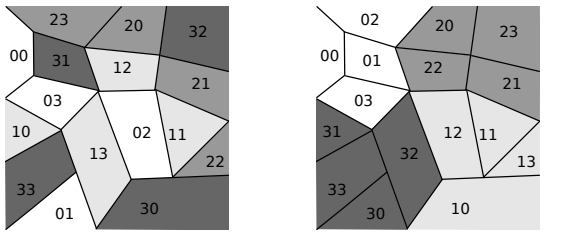

**Figure 10: Minimum tables**

the same most significant bits. We can therefore avoid storing the 4 most significant bits of the 4 first components each database vector. This saves $4 \times 4$ bits $= 16$ bits on the $8 \times 8$ bits $= 64$ bits of each vector, which leads to a 25% reduction in memory consumption. Thus, on Figure 9b, the grayed out hexadecimal digits (which represent 4 bits) may not be stored.

## 4.3 Minimum Tables

We grouped vectors to build the first four small tables $S_0, \cdots, S_3$. To build the last four small tables, $S_4, \cdots, S_7$ we compute minimum tables. This involves dividing the original distance tables, $D_4, \cdots, D_7$, into 16 portions of 16 elements each. We then keep the minimum of each portion to obtain a table of 16 elements. This process is shown on Figure 10. Using the minimum tables techniques alone results in small tables containing low values, which is detrimental to PQ Fast Scan performance. If these values are too low, the computed lower bound is not tight, i.e., far from the actual pqdistance. This limits the ability of PQ Fast Scan to prune costly pqdistance computations.

To obtain small tables with higher values, we introduce an *optimized assignment* of sub-quantizer centroids indexes. Each value $D_j[i]$ in a distance table is the distance between the $j^{\text{th}}$ sub-vector of the query vector and the centroid with index $i$ of the $j^{\text{th}}$ sub-quantizer (Section 2.2). When a sub-quantizer is learned, centroids indexes are assigned arbitrarily. Therefore, there is no specific relation between centroids having indexes corresponding to a portion of a distance table (e.g., centroids having indexes between `00` and `0f`). On the contrary, our optimized assignment ensures that all indexes corresponding to a given portion (e.g., `00` to `0f`) are assigned to centroids close to each other, as shown on Figure 11. Centroids corresponding to the same portion have the same background color. For the sake of clarity, Figure 11 shows 4 portions of 4 indexes, but in practice we have 16 portions of 16 indexes. This optimized assignment is beneficial because it is likely that a query sub-vector close to a given centroid will also be close to nearby centroids. Therefore, all values in a given portion of a distance table will be close. This allows computing minimum tables with higher values, and thus tighter lower bounds. To obtain this optimized assignment, we group centroids into 16 clusters of 16 elements each using a variant of k-means that forces groups of same sizes [24]. Centroids in the same cluster are given consecutive indexes, corresponding to one portion of a distance table. This optimized assignment of centroid indexes replaces the arbitrary assignement applied while learning sub-quantizers.




(a) Arbitrary assignment (b) Optimized assignment

**Figure 11: Centroid indexes assignement**

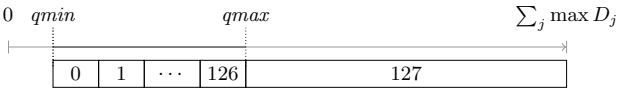



**Figure 12: Selection of quantization bounds**

## 4.4 Quantization of Distances

The vector grouping and minimum tables techniques are used to build tables of 16 elements of 32 bits each, from the original $D_j$ distance tables ($256 \times 32$ bits). So that these tables can be used as small tables, we also need to shrink each element to 8 bits. To do so, we quantize floating-point distances to 8-bit integers. As there is no SIMD instruction to compare *unsigned* 8-bit integers, we quantize distances to *signed* 8-bit integers, only utilizing their positive range, i.e., 0-127. We quantize floating-point distances between a *qmin* and a *qmax* bound into $n = 127$ bins. The size of each bin is $(qmax - qmin)/n$ and the bin number (0-126) is used as a representation value for the quantized float. All distances above *qmax* are quantized to 127 (Figure 12).

We set *qmin* to the minimum value across all distance tables, which is the smallest distance we need to represent. Setting *qmax* to the maximum possible distance, i.e., the sum of the maximums of all distance tables, results in a high quantization error. Therefore, to determine *qmax*, we find a *temporary* nearest neighbor of the query vector among the *keep*% first vectors of the database (usually, *keep* $\approx 1\%$) using the original PQ Scan algorithm. We then use the distance between the query vector and this temporary nearest neighbor as *qmax* bound. We do not need to represent distances higher than this distance because all future nearest neighbor candidates will be closer to the query vector

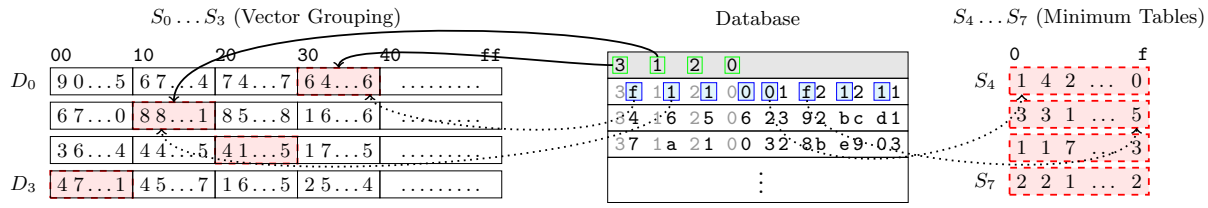



**Figure 13: Use of small tables to compute lower bounds**

than this temporary nearest neighbor. This choice of *qmin* and *qmax* bounds allows us to represent a small but relevant range of distances (Figure 12). Quantization error is therefore minimal, as confirmed by our experimental results (Section 5.5). Lastly, to avoid integer overflow issues, we use *saturated* SIMD additions.

## 4.5 Lookups in Small Tables

In this subsection, we describe how lower bounds are computed by looking up values in small tables and adding them. The first four small tables, $S_0, \cdots, S_3$ correspond to quantized portions of $D_0, \cdots, D_3$. We load these quantized portions into SIMD registers before scanning each group, as shown by the solid arrows on Figure 13. Thus, two different groups use different small tables $S_0, \cdots, S_3$. On the contrary, the last four small tables, $S_4 \cdots, S_7$, built by computing minimum tables do not change and are used to scan the whole database. They are loaded into SIMD registers at the beginning of the scan process.

As small tables contain 16 values, they are indexed by 4 bits. Given a database vector $p$, we use the 4 *least* significant bits of $p[0] \cdots p[3]$ to index values in $S_0, \cdots, S_3$ and the 4 *most* significant bits of $p[4] \cdots p[7]$ to index values in $S_4, \cdots, S_7$. Indexes are circled on Figure 13 (the 4 most significant bits correspond to the first hexadecimal digit, and the 4 least significant bits to the second hexadecimal digit) and lookups in small tables are depicted by dotted arrows. Lookups depicted by dotted arrows are performed using `pshufb`. To compute the lower bound, we add the 8 looked up values. To decide on pruning pqdistance computations, the lower bound is compared to the *quantized* value of *min*, the distance between the query vector and the current nearest neighbor.

# 5. EVALUATION

The aim of this section is twofold: evaluating the performance of PQ Fast Scan and analyzing the parameters that influence it. We show that PQ Fast Scan outperforms PQ Scan by a factor 4-6 in common usage scenarios.

## 5.1 Experimental Setup

We implemented PQ Fast Scan in C++ using intrinsics to access SIMD instructions. Our implementation uses SIMD instructions from the SSSE3, SSE3 and SSE2 instruction sets. We compared our implementation of PQ Fast Scan with the libpq implementation of PQ Scan, introduced in Section 3.1. On all our test platforms, we used the gcc and g++ compilers version 4.9.2, with the following compilation options: `-O3 -m64 -march=native -ffast-math`. We released our source code[3] under the Clear BSD license.

We evaluate PQ Fast Scan on the largest public dataset of high-dimensional vectors, ANN_SIFT1B[4]. It consists of 3 parts: a learning set of 100 million vectors, a base set of 1 billion vectors and a query set of 10000 vectors. We restricted the learning set for the product quantizer to 10 million vectors. Vectors of this dataset are SIFT descriptors of dimensionality 128. We use two subsets of ANN_SIFT1B for experiments:

- ANN_SIFT100M1, a subset of 100 million vectors of the base set. We build an index with 8 partitions; each query is directed to the most relevant partition which is then scanned with PQ Fast Scan and PQ Scan. Table 3 summarizes the sizes of the different partitions.
- ANN_SIFT1B, the full base set of 1 billion vectors to test our algorithm on a larger scale.

We study the following parameters which impact the performance of PQ Fast Scan:

- *keep*, the percentage of vectors kept at the beginning of the database (Section 4.4). Even when using PQ Fast Scan, these vectors are scanned using the seminal PQ Scan algorithm to find a temporary nearest neighbor. The distance of the query vector to the temporary nearest neighbor is then used as the *qmax* value for quantization of distance tables.
- *topk*, the number of nearest neighbors returned by the search process. For the sake of simplicity, we described PQ Scan and PQ Fast Scan as if they returned a single nearest neighbor. In practice, they return multiple nearest neighbors e.g., *topk* = 100 for information retrieval in multimedia databases.
- *partition size*, the number of vectors in the scanned partition.

**Table 3: Size of partitions used for experiments**

| Partition | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| # vectors | 25M | 3.4M | 11M | 11M | 11M | 11M | 4M | 23M |
| # queries | 2595 | 307 | 1184 | 1032 | 1139 | 1036 | 390 | 2317 |

We compare the performance of PQ Fast Scan, denoted *fastpq*, and the libpq implementation of PQ Scan, denoted *libpq*. We compare their respective *scan speed*, expressed in millions of vectors scanned per second [M vecs/s]. Scan speed is obtained by dividing response times by partition sizes. We do not evaluate PQ Fast Scan accuracy, recall or precision because PQ Fast Scan returns the exact same

[3] `https://github.com/technicolor-research/pq-fast-scan`

[4] `http://corpus-texmex.irisa.fr/`

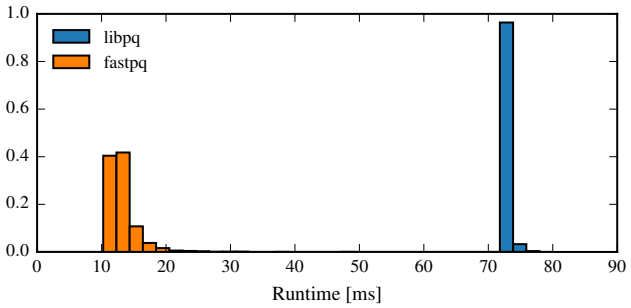



**Figure 14: Distribution of scan times (partition 0, keep=0.5%, topk=100)**

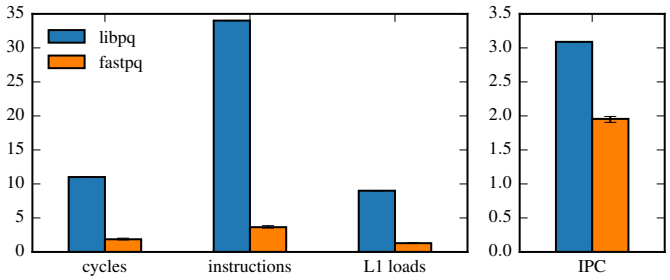



**Figure 15: Performance counters (partition 0, keep=0.5%, topk=100)**

results as PQ Scan and PQ accuracy has already been extensively studied [14]. In addition to theoretical guarantees, we checked that PQ Fast Scan returned the same results as the libpq implementation of PQ Scan for every experiment. Lastly, we run PQ Fast Scan across a variety of different platforms (Table 5) and demonstrate it consistently outperforms PQ Scan by a factor of 4-6. All experiments were run on a *single processor core*. Unless otherwise noted, experiments were run on laptop (A) (Table 5).

## 5.2 Distribution of Response Times

We study the distribution of PQ Fast Scan response times. Contrary to PQ Scan, PQ Fast Scan response time varies with the query vector. Indeed, PQ Fast Scan performance depends on the amount of pqdistance computations that can be pruned, which depends on the query vector. Figure 14 shows the distribution of response times of 2595 nearest neighbor queries executed on partition 0. As expected, PQ Scan response time is almost constant across different query vectors. PQ Fast Scan response time is more dispersed, but it responds to the bulk of queries 4-6 times faster than PQ Scan, as shown in Table 4. In the remainder of this section, when studying the impact of different parameters on PQ Fast Scan performance, we plot *median* response times or *median* scan speeds. We use the *1st quartile* (25th percentile) and *3rd quartile* (75th percentile) to draw error bars. Because it directly impacts performance, we also plot the percentage of pruned pqdistance computations.

**Table 4: Response time distribution**

| | Mean | 25% | Median | 75% | 95% |
|---|---|---|---|---|---|
| PQ Scan | 73.9 | 73.6 | 73.8 | 74.0 | 74.5 |
| PQ Fast Scan | 13.7 | 12.3 | 12.9 | 14.1 | 18.0 |
| **Speedup** | **5.4** | **6.0** | **5.7** | **5.2** | **4.1** |

## 5.3 Performance Counters

We use performance counters to measure the usage of CPU resources of PQ Fast Scan and PQ Scan when scanning partition 0 (Figure 15). Thanks to the use of register-resident small tables, PQ Fast Scan only performs 1.3 L1 loads per scanned vector, where the libpq implementation of PQ Scan requires 9 L1 loads. PQ Fast Scan requires 89% less instructions than PQ Scan thanks to the use of SIMD instructions instead of scalar ones (respectively 3.7 and 34 instructions per scanned vector). PQ Fast Scan uses 83% less cycles than PQ Scan (respectively 1.9 and 11 cycles per vector). The decrease in cycles is slightly less significant than the decrease in instructions because PQ Fast Scan has a lower IPC than PQ Scan. This is because SIMD instructions can be less easily pipelined than scalar instructions.

## 5.4 Impact of keep and topk Parameters

Both *keep* and *topk* impact the amount of pruned distance computations, and thus PQ Fast Scan performance. For information retrieval in multimedia databases, *topk* is often set between 100 and 1000. Therefore, we start by studying the impact of *keep* for $topk = 100$ and $topk = 1000$. The *keep* parameter impacts the tightness of the *qmax* bound used for quantization. A higher *keep* value means more vectors are scanned using the seminal PQ Scan algorithm to find a temporary nearest neighbor (Section 4.4). This makes the *qmax* bound tighter and decreases the distance quantization error. Figure 16 shows that the pruning power increases with *keep*; however this increase is moderate. For $topk = 1000$, the pruning power is lower than for $topk = 100$ and more sensitive to *keep*. PQ Fast Scan can prune a pqdistance computation if the lower bound of the currently scanned vector is higher than the distance between the query vector and the current topk-th nearest neighbor. A higher *topk* value implies a higher distance between the query vector and the topk-th nearest neighbor. Therefore, less pqdistance computations can be pruned.

The scan speed increases slightly with *keep* as more distance computations get pruned, up to a threshold where it starts to collapse. After this threshold, the increase in pruned distance computations provided by the tighter *qmax* bound is outweighed by the increased time spent scanning the first *keep*% vectors using the slow PQ Scan algorithm. Overall, PQ Fast Scan is not very sensitive to *keep*, and a decent *qmax* bound is found quickly. Any *keep* value between 0.1% and 1% is suitable. We set $keep = 0.5\%$ for the remainder of experiments. Lastly, we evaluate PQ Fast Scan performance for more *topk* values. Figure 18 confirms that PQ Fast Scan performance decreases with *topk*.

## 5.5 Impact of Distance Quantization

PQ Fast Scan uses three techniques to build small tables: (1) vector grouping, (2) minimum tables and (3) quantization of distances. Among these three techniques, minimum tables and quantization of distances impact the tightness of

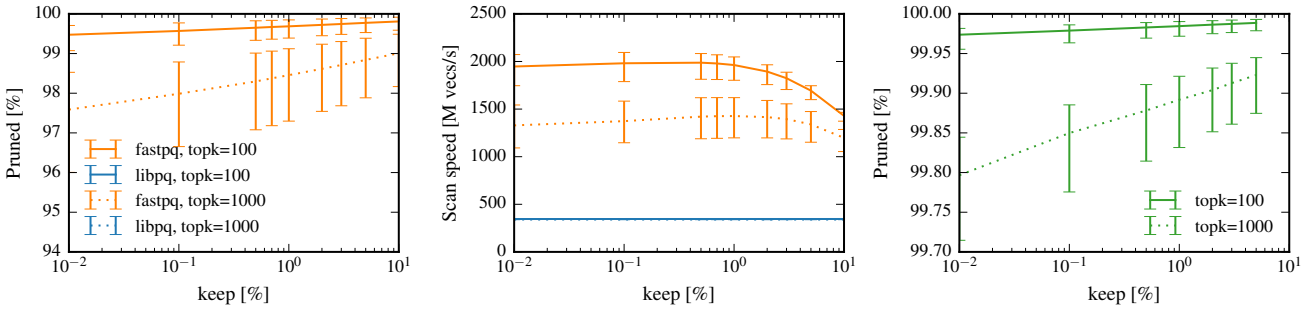



**Figure 16: Impact of keep parameter (all partitions)**

**Figure 17: Pruning power using quantization only (all partitions)**

lower bounds, and therefore pruning power. To assess the respective impact on pruning power of these two techniques, we implement a quantization-only version of PQ Fast Scan which relies *only* on quantization of distances (Figure 17). This version uses tables of 256 8-bit integers, while the full version of PQ Fast Scan uses tables of 16 8-bit integers. Therefore, the quantization-only version cannot use SIMD and offers no speedup. Hence, Figure 17 shows only the pruning power and does not show the scan speed. The quantization-only version of PQ Fast Scan achieves 99.9% to 99.97% pruning power. This is higher than the pruning power of the full version of PQ Fast Scan (i.e., using the three techniques) which is 98% to 99.7% (Figure 16). This demonstrates that our quantization scheme is highly efficient and that most of the loss of pruning power comes from minimum tables.

## 5.6 Impact of Partition Size

The partition size impacts scan speed without impacting pruning power (Figure 19). Partitions 0, 7, 2, 4, 5 and 3 have sizes comprised between 10 million vectors and 25 million vectors, and PQ Fast Scan speed is almost constant across all theses partitions. Smaller partitions, e.g., partitions 6 and 1, exihibit lower scan speeds. PQ Fast Scan groups vectors on 4 components for partition exceeding 3 million vectors, (Section 4.2). As partition sizes approaches this threshold, the scan speed decreases. For partitions comprising less than 3 million vectors, it would be necessary to group vectors on 3 components instead of 4.

## 5.7 Large Scale Experiment

We test PQ Fast Scan on the full database of 1 billion vectors (ANN_SIFT1B). For this database, we build an index with 128 partitions. Partitions therefore have an average size of about 8 million vectors. We run 10000 NN queries. The most appropriate partition for each query is selected using the index, and scanned to find nearest neighbors. We scan partitions using both PQ Scan and PQ Fast Scan, and we compare mean response times to queries (Figure 20, SIFT1B). In addition to its lower response time, PQ Fast Scan also allows decreasing the amount of memory consumed by the database thanks to vector grouping (Section 4.2). Unlike previous experiments, this experiment was run on workstation (B) instead of laptop (A) (Table 5). The parameters $keep = 1\%$, $topk = 100$ were chosen.

**Table 5: Configuration of test platforms**

| laptop (A) | workstation (B) | server (C) | server (D) |
|---|---|---|---|
| Core<br>i7-4810MQ<br>Haswell<br>2.8-3.8 Ghz<br>2014 | Xeon<br>E5-2609v2<br>Ivy Bridge<br>2.5-2.5 Ghz<br>2013 | Xeon<br>E5-2640<br>Sandy Bridge<br>2.5-3.0 Ghz<br>2012 | Xeon<br>X5570<br>Nehalem<br>2.9-3.3 Ghz<br>2009 |
| 8 GB<br>2×4 GB<br>DDR3<br>1600 Mhz | 16 GB<br>4×4 GB<br>DDR3<br>1333 Mhz | 64 GB<br>4×16 GB<br>DDR3<br>1333 Mhz | 24 GB<br>6×4 GB<br>DDR3<br>1066 Mhz |

## 5.8 Impact of CPU Architecture

To conclude our evaluation section, we compare PQ Fast Scan and PQ Scan over a wide range of using processors released between 2009 and 2014 (Table 5). On all these systems, Fast PQ Scan median speed exceeds PQ Scan median speed by a factor of 4-6, thus validating our performance analysis and design hypotheses (Figure 20, Scan speed). PQ Fast Scan performance is not sensitive to processor architecture. PQ Fast Scan loads 6 bytes from memory for each lower bound computation. Thus, a scan speed of 1800 M vecs/s correspond to a bandwith use of 10.8 GB/s. The memory bandwidth of Intel server processors ranges from 40 GB/s to 70 GB/s. When answering 8 queries concurrently on an 8-core server processor, PQ Fast Scan is bound by the memory bandwidth, thus demonstrating its highly efficient use of CPU resources.

# 6. DISCUSSION

While this paper mainly focuses on ANN search, the techniques used in PQ Fast Scan can be applied beyond this context. We now discuss how they can be generalized.

The main idea behind PQ Fast Scan is to build lookup tables so that they fit in SIMD registers, while storing them in the L1 cache is generally considered as best practice for efficiency. Therefore, any algorithm relying on lookup tables is a candidate for applying this idea. Among practical uses of lookup tables is query execution in compressed databases. Compression schemes, either generic [23, 12, 3, 25] or specific (e.g., SAX for time series [18]), have been widely adopted in database systems. In the case of dictionary-based compression (or quantization), the database stores compact codes. A dictionary (or codebook) holds the actual values corresponding to the compact codes. Query execution then relies

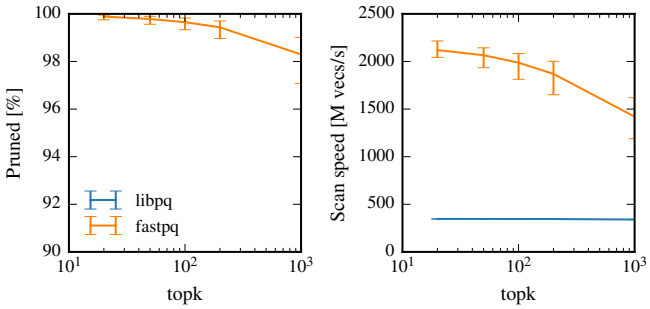



**Figure 18: Impact of topk parameter (all partitions, keep=0.5%)**

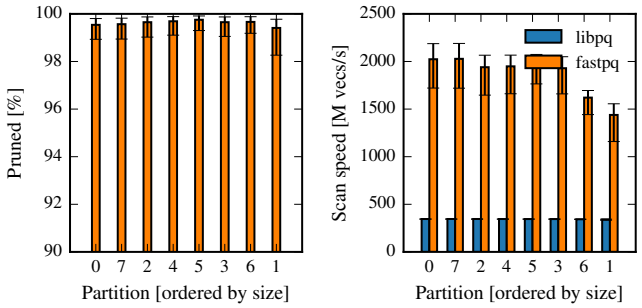



**Figure 19: Impact of partition size (all partitions, keep=0.5%, topk=100)**

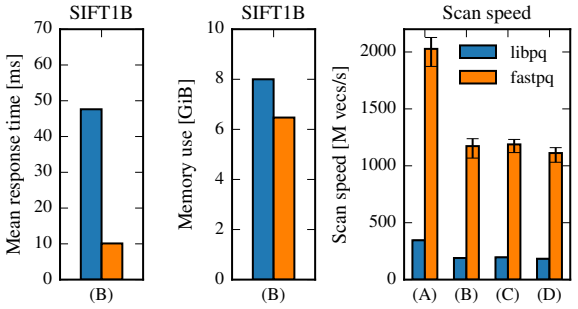



**Figure 20: Experiments on different platforms (see Table 5)**

on lookup tables, derived from the dictionary. In this case, storing lookup tables in SIMD registers allows for better performance. If lookup tables are small enough (16 entries), they may be stored directly in SIMD registers, after quantization of their elements to 8-bit integers. Otherwise, it possible to build small tables for different types of queries. For top-k queries, it is possible to build small tables enabling computation of lower or upper bounds. Like in PQ Fast Scan, lower bounds can then be used to limit L1-cache accesses. To compute upper bounds instead of lower bounds, maximum tables can be used instead of minimum tables. For approximate aggregate queries (e.g., approximate mean), tables of aggregates (e.g., tables of means) can be used instead of minimum tables.

Another idea behind PQ Scan Fast is to use 8-bit saturated arithmetic. This idea can be applied for queries which do not use lookup tables, such as queries executed on uncompressed data. Top-k queries require exact score evaluation for a small number of items, so 8-bit arithmetic can be used to discard candidates. Similarly, 8-bit arithmetic may provide enough precision for approximate queries. In the context of SIMD processing, 8-bit arithmetic allows processing 4 times more data per instruction than 32-bit floating-point arithmetic and thus provides a significant speedup.

To perform lookups in tables stored in SIMD registers, we use SIMD shuffle instructions (included in the Intel SSSE3 instruction set). Such instructions are also available on ARM processors, with the Neon instruction set. Lastly, the AVX-512 SIMD instruction set, available on upcoming Intel processors, will allow storing larger tables in SIMD registers. This will allow for even better performance and wider applicability of our techniques.

# 7. RELATED WORK

*Product Quantization for ANN.* Product Quantization is a widely used solution for ANN search in large databases. Several publications extend the original method presented in [14]. Among these publications, a part focuses on the development of efficient indexing schemes that can be used in conjunction with product quantization [4, 28]. Another part of publications focuses on optimizing the learning of sub-quantizers to increase recall [21, 10, 15], which is orthogonal to our work. Adaptation of PQ Fast Scan to these optimized product quantizers is straightforward, as they also rely on distance tables for ANN search.

*Other approaches for ANN.* Locality Sensitive Hashing (LSH) is another prominent approach for ANN search. The original method [8] has prohibitive storage requirements for large databases but offers theoretical guarantees on the quality of the returned neighbors. LSH-based systems with lower storage requirements have been recently proposed [9, 19]. Despite recent improvements, LSH-based systems have higher storage requirements than systems based on product quantization. LSH-based systems are therefore less suited to store very large databases in main memory.

*Lookup tables in SIMD registers.* Implementation of erasure correcting codes, used to provide reliable storage, relies on cache-resident lookup tables. Authors have proposed to shrink these tables and store them into SIMD registers [22, 17]. This allowed a dramatic improvement in throughput over previous approaches. These lookup tables are used to implement associative finite field arithmetic, and these associativity properties can be used to shrink lookup tables. In PQ Scan, distance computations are not associative, so we developed other techniques to shrink lookup tables.

*Fast query processing.* Operating directly on compressed data (or compact representations), like in PQ Fast Scan, has been shown to speed up query processing in other scenarios [3, 25]. Besides, the use of SIMD to speed up query processing in databases [6] has been widely studied. In particular, SIMD has been used to sort data [7] and perform relational joins [5, 16]. The specificity of our work is to focus on use of SIMD to speed up algorithms relying on lookup tables, such as PQ Scan.

# 8. CONCLUSION

In this paper, we presented PQ Fast Scan, a novel algorithm for ANN search that is able to check over a billion candidate vectors per second on a single core. PQ Fast Scan builds on product quantization, a recently introduced and increasingly popular method for ANN search in high-dimensional spaces. PQ Fast Scan design stems from a thorough analysis of the limitations of PQ Scan, the original algorithm for product quantization based ANN search.

An important feature of PQ Scan is that it relies on L1-cache resident lookup tables to compute distances. Although using cache resident lookup tables is generally seen as sufficient for efficiency, we demonstrated that storing lookup tables in SIMD registers allows achieving significantly lower query response time. Our main contribution lies in the design of techniques to turn cache-resident lookup tables into small tables, sized to fit SIMD registers. Use of these small tables allow PQ Fast Scan to perform 4-6 times faster than PQ Scan, thus advancing the state of art in high-dimensional ANN search.

## Acknowledgements

We thank the anonymous reviewers for their suggestions to improve this paper. Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see `https://www.grid5000.fr`).

# 9. REFERENCES

[1] Intel® 64 and IA-32 Architectures Optimization Reference Manual, April 2012.

[2] Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture, June 2015.

[3] D. J. Abadi, S. R. Madden, and M. C. Ferreira. Integrating Compression and Execution in Column-Oriented Database Systems. In *SIGMOD*, pages 671–682, 2006.

[4] A. Babenko and V. Lempitsky. The Inverted Multi-Index. In *CVPR*, pages 3069–3076, 2012.

[5] C. Balkesen, G. Alonso, J. Teubner, and M. T. Ozsu. Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited. *PVLDB*, 7(1):85–96, 2013.

[6] P. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*, pages 225–237, 2005.

[7] J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y.-K. Chen, A. Baransi, S. Kumar, and P. Dubey. Efficient implementation of sorting on multi-core SIMD CPU architecture. *PVLDB*, 1(2):1313–1324, 2008.

[8] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *SCG*, pages 253–262. ACM, 2004.

[9] J. Gan, J. Feng, Q. Fang, and W. Ng. Locality-sensitive Hashing Scheme Based on Dynamic Collision Counting. In *SIGMOD*, pages 541–552, 2012.

[10] T. Ge, K. He, Q. Ke, and J. Sun. Optimized Product Quantization. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 36(4):744–755, 2014.

[11] A. Gionis, P. Indyk, and R. Motwani. Similarity Search in High Dimensions via Hashing. In *VLDB*, pages 518–529, 1999.

[12] G. Graefe and L. D. Shapiro. Data compression and database performance. In *SAC*, pages 22–27, 1991.

[13] J. Hofmann, J. Treibig, G. Hager, and G. Wellein. Comparing the Performance of Different x86 SIMD Instruction Sets for a Medical Imaging Application on Modern Multi- and Manycore Chips. In *WPMVP*, pages 57–64. ACM, 2014.

[14] H. Jégou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(1):117–28, 2011.

[15] Y. Kalantidis and Y. Avrithis. Locally Optimized Product Quantization for Approximate Nearest Neighbor Search. In *CVPR*, pages 2329–2336, 2014.

[16] C. Kim, T. Kaldewey, V. W. V. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. Di Blas, and P. Dubey. Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-core CPUs. *PVLDB*, 2(2):1378–1389, 2009.

[17] H. Li and Q. Huan-yan. Parallelized Network Coding with SIMD Instruction Sets. In *ISCSCT*, volume 1, pages 364–369, 2008.

[18] J. Lin, E. Keogh, L. Wei, and S. Lonardi. Experiencing SAX: a novel symbolic representation of time series. *Data Mining and Knowledge Discovery*, 15(2):107–144, 2007.

[19] Y. Liu, J. Cui, Z. Huang, H. Li, and H. Shen. SK-LSH: An Efficient Index Structure for Approximate Nearest Neighbor Search. *PVLDB*, 7(9):745–756, 2014.

[20] S. Lloyd. Least squares quantization in PCM. *IEEE Transactions on Information Theory*, 28(2):129–137, 1982.

[21] M. Norouzi and D. J. Fleet. Cartesian K-Means. In *CVPR*, pages 3017–3024, 2013.

[22] J. S. Plank, K. Greenan, and E. L. Miller. Screaming Fast Galois Field Arithmetic Using Intel SIMD Extensions. In *FAST*, pages 298–306, 2013.

[23] M. A. Roth and S. J. Van Horn. Database compression. *ACM Sigmod Record*, 22(3):31–39, 1993.

[24] E. Schubert. Same-size k-means variation, 2012. `http://elki.dbs.ifi.lmu.de/wiki/Tutorial/SameSizeKMeans`.

[25] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, et al. C-store: a column-oriented DBMS. In *VLDB*, pages 553–564, 2005.

[26] Y. Sun, W. Wang, J. Qin, Y. Zhang, and X. Lin. SRS : Solving c -Approximate Nearest Neighbor Queries in High Dimensional Euclidean Space with a Tiny Index. *PVLDB*, 8(1):1–12, 2014.

[27] R. Tavenard, H. Jegou, M. Douze, and L. Amsaleg. Searching in one billion vectors: Re-rank with source coding. In *ICASSP*, pages 861–864, 2011.

[28] Y. Xia, K. He, F. Wen, and J. Sun. Joint Inverted Indexing. In *ICCV*, pages 3416–3423, 2013.