

# Impact of Storage Technology on the Efficiency of Cluster-Based High-Dimensional Index Creation

Gylfi Þór Gudmundsson<sup>1</sup>, Laurent Amsaleg<sup>1,2</sup>, and Björn Þór Jónsson<sup>3</sup>

<sup>1</sup> INRIA, Rennes, France

{gylfi.gudmundsson, laurent.amsaleg}@inria.fr

<sup>2</sup> CNRS, Rennes, France

<sup>3</sup> School of Computer Science, Reykjavík University, Iceland  
bjorn@ru.is

**Abstract.** The scale of multimedia data collections is expanding at a very fast rate. In order to cope with this growth, the high-dimensional indexing methods used for content-based multimedia retrieval must adapt gracefully to secondary storage. Recent progress in storage technology, however, means that algorithm designers must now cope with a spectrum of secondary storage solutions, ranging from traditional magnetic hard drives to state-of-the-art solid state disks. We study the impact of storage technology on a simple, prototypical high-dimensional indexing method for large scale query processing. We show that while the algorithm implementation deeply impacts the performance of the indexing method, the choice of underlying storage technology is equally important.

## 1 Introduction

Multimedia data collections are now extremely large and algorithms performing content-based retrieval must deal with secondary storage. Magnetic disks have been around for decades, but their performance, aside from capacity, has not improved significantly. Better storage performance, and improved reliability, has been achieved, however, by grouping many disks together and striping data as in the Redundant Array of Independent Disks approach. Recently, Solid-State Disks (SSDs) have emerged as a disruptive storage technique based on memory cells on chips. Their storage capacity grows quickly and they outperform magnetic approaches. It is therefore of high interest to study what impact secondary storage technologies can have on the design and performance of high-dimensional indexing algorithms that are a core component of content-based retrieval approaches. This paper is an initial investigation in that direction.

### 1.1 Content-Based Retrieval and High-Dimensional Indexing

Retrieving multimedia documents based on their *content* means that the search analyzes the actual content of the documents rather than metadata associated with the documents, such as keywords, tags, and free text descriptions [11]. For images, the term content might, for example, refer to colors, shapes, texture, or

any other information, possibly very fine-grained, that can be derived from the image itself [7]. Users can then submit a photo to the system to search for multimedia material that is visually similar. For music, information such as pitch, energy and timbral features can be used for such content-based retrieval. Searching by content is needed for such applications as image copyright enforcement, face recognition, query by humming, or automatic image classification [5].

Content-based retrieval systems are frequently built from two primary software components: the first component automatically extracts some low level features from the multimedia material; while the second component builds an index over this database of features allowing for efficient search and retrieval. For images, the computer vision literature includes numerous feature extraction techniques. The visual properties of each image are encoded as a set of numerical values, which define high-dimensional vectors. There is one such vector per image when the description is global, for example when a color histogram encodes the color diversity of the entire image. Some applications require a much more fine-grained description; in this case multiple features are extracted from small regions in each image [12]. Typically, the texture and/or the shape that are observed in each region of interest are somehow encoded in a vector. Overall, the similarity between images is determined by computing the distance between the vector(s) extracted from the query image and the ones kept in the database. As vectors are high dimensional, repeatedly executing this distance function (typically Euclidean) to run a *k-nearest neighbor* search is CPU intensive.

Efficient retrieval is facilitated by high-dimensional indexing techniques that prune the search space (e.g, see [15]). Most techniques from the abundant literature partition the database of features extracted from the collection of images into cells and maintain a tree of cell representatives. The search then takes a query vector, traverses the tree according to the closest representative at each level, fetches bottom leafs containing database features, computes distances and returns the *k* nearest-neighbors found. At a large scale, when indexing a few hundred millions vectors or more, approximate search schemes returning near-neighbors (possibly not the nearest) must be used for efficiency, potentially trading result quality for response time [1, 6, 10].

## 1.2 Contribution

We have created two very different implementations of a rather simple, yet very effective, high-dimensional indexing strategy relying on clustering to partition the database of features [8]. These implementations differ in the way they access secondary storage, emphasizing small vs. large I/Os and random vs. sequential I/Os. We have then run these implementations on a machine connected to various magnetic storage devices, as well as various SSD devices, and accurately logged their respective performance.

This study focuses on the efficiency of the index creation, which is the most disk-intensive phase, far more intensive than the search phase. Not only must the entire data collection be read from secondary storage during index creation, and then eventually written back to secondary storage again, but a gigantic

number of CPU intensive distance calculations between vectors are also required to cluster them. High-dimensional indexes are typically created in a bulk manner: all vectors to index are known before the process starts and the index tree structure, as well as the bottom leaves of the tree, are all created in one go. From a traditional DBMS perspective, this process can be seen as being analogous to a sort-merge process with very CPU-intensive comparison function calls.

With our detailed analysis, we show that a good understanding of the underlying hardware is required at design time to get optimal performance. In particular, as the devil is in the details, we show that the theoretical behavior of storage devices may differ much from what is observed in the battlefield and that even members of a presumably homogeneous family of storage solutions may behave quite differently. We also show that balancing efforts on software versus hardware is not always obvious: it is key to clearly evaluate the cost of paying a very skilled programmer to nicely tune and debug a complex piece of code versus producing a naive implementation and throwing at it efficient hardware.

### 1.3 The Case for Reality

This investigation is done using real algorithms working on real data with standard production hardware. Working with various technical specifications does not accurately reflect the complex layers of interactions between application, operating system and I/O devices (disks, network) while processing real data.

We implemented a cluster-based high-dimensional indexing algorithm that is prototypical of many approaches published in the literature. The algorithm is approximate in order to cope with truly large high-dimensional collections. It has an initial off-line phase that builds an index. This is a very demanding process: the entire raw data collection is read from disk and vectors that are close in the feature space are clustered together and then written back to disk. This process is essentially I/O bound.

The search phase is quite different, but it is also prototypical of what has been published. From a query submitted by a user, a few candidate clusters are identified using the index, fetched from disks, and then the CPU is heavily used for scanning the clusters in search of similar vectors.

We also use a real data collection made of more than 110 million local SIFT descriptors [12] computed on real images randomly downloaded from Flickr. This descriptor collection is clearly not made of independent and identically distributed random variables as is the case for most synthetic benchmarks; it has some very dense areas, while some others are very sparse, which together strongly challenge the indexing and retrieval algorithms. Observing the behavior of indexing and searching real data is known to give more valuable insights in general [16], and this certainly extends to the impact of various storage solutions.

### 1.4 Overview of the Paper

This paper is structured as follows. Section 2 briefly reviews the state of art in secondary storage techniques. Section 3 gives an overview of the high dimen-

sional indexing algorithm we use in this paper, while Section 4 details the two implementations which stress differently the underlying hardware. Section 5 then describes the experiments we performed and Section 6 concludes the paper.

## 2 Secondary Storage Review

We now briefly review the state-of-the-art in storage technology, focusing on the aspects that are most relevant for our work.

### 2.1 Magnetic Disks

Magnetic disks are the standard for cheap secondary storage. During the last decade, only the capacity of the disks has dramatically increased; the latest is the terabyte platter announced by Seagate in May 2011. Their read/write performance has improved little as their mechanical parts are inherently slow. This impacts the performance of small and random operations which are significantly slower than large sequential ones. Accessing data that is contiguous on disk is therefore key to disk performance. Note that sophisticated embedded software in the disk controller tries to minimize the costs of reading and writing (e.g., reordering accesses, enforcing contiguity, writing asynchronously, caching) but programmers have little or no control over these decisions.

### 2.2 Solid-State Disks

SSD technology is based on flash memory chips. Two types of memory cells are used: single-level chips (SLC), which are fast and durable, and multi-level chips (MLC) that take more space and are not as durable, but are cheaper. Recently Intel, with its 710 Series SSD, introduced new MLC technology that is nearly on par with SLC in endurance. This new SSD is targeting the needs of the enterprise with both endurance and capacity.

The memory modules are typically arranged in 128KB blocks. Since there are no slow mechanical parts, reading from an SSD is extremely fast and sequential or random reads are equally fast. In contrast, writing is more costly as it sometimes requires a special erase operation done at the level of an entire 128KB block. The cost of writes is therefore not uniform and write performance is typically unpredictable from a programmer's point of view. In addition to minimizing write costs, internal controlling algorithms do wear-leveling to extend the life span of the chips. SSD performance has been extensively studied (e.g., see [4]).

With the release of the SATAIII standard, the potential transfer rate doubled, from 300MB/sec to 600MB/sec. In turn, the SSD vendors released 500+MB/sec. capable disks for the public market. The enterprise market, however, has shown more restraint in this area and focused on durability and capacity. For example, the Intel 710 Series disks have only 270MB/s read capability, and 170MB/s write capability, far below the capacity of SATAIII.

### 2.3 Hybrid Disk: Magnetic Disk and SSD in One Device

Seamlessly combining SSD and magnetic disk technology into a single device has long been expected. The first such disk, Momentus XT, was introduced by Seagate in 2010. It is basically a 250-500GB 7200rpm magnetic disk that has a single 4GB SLC cell embedded. However, as there is only one SSD cell, the read/write capacity is limited. Furthermore, the SSD cell is only used for read caching. With such limitations, the Momentus XT is primarily targeting the laptop market where it provides power savings and reduces boot- and loading time. One can fully expect, however, to see rapid advances in this area.

### 2.4 Network Attached Storage (NAS)

A NAS is typically a quite large secondary storage solution made available over a network. It usually contain an array of disks, operating in parallel thanks to advanced RAID controllers, and made available through a network connected file-servers or dedicated hardware directly connected to the network. The performance of the NAS can be very hard to evaluate as there are many layers of hardware, caching and communication, each with its own bottlenecks. Often the network links between the server and the clients limit the throughput as the links are typically shared by many clients.

### 2.5 Interactions with the OS

Many sophisticated routines trying to reduce the costs of accessing secondary storage exist in all operating systems. Prefetching is a common technique: instead of reading few bytes at a time from the disk, more data than asked for is brought into RAM in the hope that data needed later will therefore already be in memory, thus avoiding subsequent disk accesses. Reads are blocking operations and the requesting process can only be resumed once the data is in memory, but writes might be handled asynchronously as there is effectively no need to wait for the data to reach the disk. Overall, the operating system uses buffers for I/Os and fills or flushes them when it so desires, trying to overlap the I/O and CPU load as much as possible. If reads or writes are issued too rapidly, there is little overlapping and performance degrades.

## 3 Extended Cluster Pruning

To study the impact of secondary storage technologies on high-dimensional indexing, we implemented a prototypical index creation scheme built on the Cluster Pruning algorithm [6], as extended by [8]. Cluster Pruning is quite representative of the core principles underpinning many of the quantification-based high-dimensional indexing algorithms that perform very well [17, 14, 9].

Overall, the extended Cluster Pruning algorithm (eCP) is very much related to the well-known  $k$ -means approach. Like  $k$ -means, eCP is an unstructured

quantifier, thus coping quite well with the true distribution of data in the high-dimensional space [13]. The extension in eCP is to make the algorithm more I/O friendly as the database is assumed to be too large to fit in memory and must therefore reside on secondary storage.

### 3.1 Index Construction

eCP starts by randomly selecting  $C$  points from the data collection, which are used as representatives of the  $C$  clusters that eCP will eventually build. The cluster count  $C$  is typically chosen such that the average cluster size will fit within one disk I/O, which is key to secondary storage performance. Then the remaining points from the data collections are read, one after the other, and assigned to the closest cluster representative (resulting in Voronoi cells).

When the data collection is large, the representatives are organized in a multi-level hierarchy. This accelerates the assignment step as finding the representative that is closest to a point then has logarithmic complexity (instead of linear complexity). Eventually, once all the raw collection has been processed, then eCP has created  $C$  clusters stored sequentially on disks, as well as a tree of representatives which is also kept on disks.

The tree structure is extremely small compared to the clusters as only a few nodes are required to form that tree. The tree is built according to a hierarchical clustering principle where the points used at each level of the tree are representatives of the points stored at the level below. This does not, however, provide total ordering of the clusters and thus this is not a B<sup>+</sup>-tree-like index.

### 3.2 Searching the Index

When searching, the query point is compared to the nodes in the tree structure to find the closest cluster representative. Then the corresponding cluster is accessed from the disk, fetched into memory, and the distances between the query point and all the points in that cluster are computed to get the  $k$  nearest neighbors.

The search is approximate as some of the true nearest neighbors may be outside the Voronoi cell under scrutiny. Experience from different application domains has shown that the quality results of eCP can be improved by searching in more than one cell, because this returns better neighbors. In this case, however, more cells must be fetched from secondary storage and more distance computations performed.

### 3.3 Result Quality of eCP

Experiments have shown shows that eCP returns good quality results despite its approximate behavior [6,8]. Two main reasons can explain this. First, the high-dimensional indexing process produces Voronoi cells that nicely preserve the notion of neighborhood in the feature space. The search process is therefore likely to find the actual nearest neighbors in the clusters identified at query

time. Second, most modern image recognition techniques use local descriptions of images where a single image is typically described using several hundred high-dimensional vectors. At search time, the many vectors extracted from the query image are all used one after the other to probe the index and get back  $k$  neighbors. What is returned for every single query vector is eventually aggregated (typically by voting) to identify the most similar images. Because there is so much redundancy in the description, missing the correct neighbors for some of the query vectors has indeed little impact on the final result.

The cluster count, and the corresponding average cluster size, obviously influences heavily the CPU cost for the search phase as the number of distance calculations is linked to the cardinality of clusters. It also impacts the performance of the index creation as it determines the total number of clusters that must be created to hold the entire collection, and thus influences the number of nodes in the index tree as well as its height and width. This, in turn, impacts the number of distance calculations done to find the cluster into which each point has to be assigned. Experiments with various cluster sizes indicate that using 64-128KB as the average cluster size gives the best performance overall [8].

## 4 Index Creation Policies

We have designed two implementations of the eCP index creation algorithm that have quite different access patterns to secondary storage. They differ during their *assignment phase*, when assigning vectors to their clusters, and also during their *merging phase*, when forming the final file that is used during the subsequent searches. We have not changed the search process of eCP at all.

Both index creation policies start by building their in-memory index tree by picking leaders from the raw collection. Then they allocate a buffer, called *in-buff*, for reading the raw data collection in large pieces. They then iterate through the raw collection via this buffer, filling it with many not-yet-indexed vectors. The index is used to quickly identify the leader that is the closest to each vector in *in-buff*, representing the cluster that the vector must be assigned to. Once all vectors in *in-buff* have been processed, one of the two policies described below is used to transfer the contents of the buffer to secondary storage.

### 4.1 Policy 1: TempFiles (TF)

This first policy uses temporary files, one for each cluster. Each temporary file contains all the vectors assigned so far to that cluster. When called, the TF policy loops through the representatives, appending to each temporary file all vectors in *in-buff* assigned to that cluster. When appending to a cluster, its associated temporary file is opened, appended to and closed, as they are too numerous to remain open. When all vectors from *in-buff* have been written to disk, a new large piece from the raw collection is read into *in-buff*, and eCP continues. After having assigned all vectors from the raw collection, all these temporary cluster

Disk	Type	Specified Ave. Seek Time	Specified Ave. Rot. Latency	Specified Cache Size	Measured Seq. R/W Thr.put.
Seagate	Magnetic	11.0 ms	4.16 ms	8 MB	46/40 MB/s
Fujitsu	Magnetic	11.5 ms	4.17 ms	16 MB	68/53 MB/s
SuperTalent	SSD	<1 ms	-	Unknown	124/34 MB/s
Intel	SSD	<1 ms	-	16 MB	220/66 MB/s

**Table 1.** Key storage device performance indicators.

files are then concatenated into a single final file by reading them sequentially from disk and writing to the final file.

In terms of access patterns, TF performs, at cluster assignment time, large sequential reads to fill *in-buff* with new vectors as well as many small random writes, one per cluster, every time all the vectors in *in-buff* have been processed. When creating the final file, it also performs cluster-sized sequential reads (one per cluster, typically 128KB) as well as large sequential writes for the final file.

## 4.2 Policy 2: ChunkFiles (CF)

This second policy generally follows a sort-merge principle. When called, CF sorts *in-buff* on increasing values of the leader identifiers. It then creates a new chunk file on disk and flushes *in-buff* into that chunk file before closing it. It then reads another large piece from the raw collection into *in-buff* and continues. After having processed all vectors from the raw collection, CF merges all the sorted chunk files using a typical secondary storage merging process.

In terms of access patterns, CF performs, at cluster assignment time, large sequential reads (typically 128MB) to fill *in-buff* and large sequential writes when creating each chunk file. When creating the final file, it performs many small random reads to get data from all the chunk files as needed and large sequential writes for the final file.

# 5 Experiments

## 5.1 Experimental Setup

In our experiments we used a collection of more than 110 million SIFT descriptors [12] of 128 dimensions extracted from 100,000 images randomly downloaded from Flickr. This collection is about 14.5GB. We reused the parameters from [8] that were found to work best, i.e., the depth of the index was 3 and the average cluster size was 128KB, resulting in 111,424 clusters on secondary storage. Note that clusters are not equally filled as the true distribution of vectors in space is not balanced (30% of the clusters are smaller than 64KB, while 21% are larger than 192KB). In all experiments the size of *in-buff*, and thus each chunk file, is 128MB. Note that this is much larger than the cluster size.

Experiments were run on a Dell Precision T3400, 3GHz Intel E6400 dual core CPU with 6MB cache and 4GB RAM (only one core was used). For all

Disk	Total Time		Assignment		Merging	
	I/O+CPU (s)		I/O (s)		I/O (s)	
	TF	CF	TF	CF	TF	CF
Seagate	43,144	12,949	12,556	548	18,829	1,299
Fujitsu	32,895	12,689	9,975	388	11,145	1,207
SuperTalent	32,540	11,528	17,120	149	3,529	236
Intel	14,164	11,398	2,028	46	402	244
NAS	22,335	14,564	5,314	610	2,349	202

**Table 2.** Performance of eCP index creation policies, single drive setups.

disks we use the `ext3` file system and Debian OS. We tested two magnetic disks: 3.5" Seagate Barracuda 7200.10 and 2.5" Fujitsu MHZ2160BJ. Both are 7200 rpm disks with similar seek time and rotational latency. We also used two SSDs: SuperTalent FTM28GL25H and Intel X-25M, type SSDSA2MH080G1GC. Finally, we used a NAS 3070 from NetApp. Table 1 provides more details on the single drives. The three first columns are filled using vendor figures, while the last column shows sustained observed sequential read and write performance. Accurately measuring the performance of the NAS is much more complicated.

We then ran two different experiments. In the first experiment we used a single drive: the file containing the raw collection, the temporary files/chunk files, and the final cluster file were all stored on a single disk. In this case some reads and writes overlap in time and compete for the disk. This causes slower performance as enforcing truly sequential accesses is much more difficult.

In the second experiment we used two drives. In this case, the raw collection was kept on one drive and the temporary files/chunk files were stored on another drive, eliminating any competition between reads and writes at assignment time. Similarly, the final file and the temporary files/chunk files were stored on different drives; it is sufficient to put the final file on the first drive where the raw collection is to eliminate any competition at merging time. We now detail the performance measurements for these two experiments.

## 5.2 Single Drive Experiment

Table 2 shows the performance measurements when using the single drive setup. The total (wall clock) time includes the time for I/Os as well as for executing the many distance calculations on the CPU. The CPU usage is almost identical for both TF and CF and equal to 11,000 seconds on average, divided into 10,930 seconds for assignment and 70 seconds for merging. The second and third columns show the overall time it takes to perform the assignment of vectors to leaders and the final merging. These times include the time spent on I/Os but exclude the almost constant CPU costs.

Overall, focusing on the total time, regardless of the device, the first key observation is that CF always outperforms TF. The TF policy repeatedly opens, writes to, and then closes clusters, forcing the OS to flush data on disks using synchronized blocking writes. TF appends data to many relatively small files (111,424 files of 128KB), in contrast to CF which writes only once to each of

Two Drive Setup	Total Time		Assignment		Merging	
	I/O+CPU (s)		I/O (s)		I/O (s)	
	TF	CF	TF	CF	TF	CF
Fujitsu-Intel-Fujitsu	13,467	11,640	1,977	370	208	220
Intel-Intel'-Intel	13,484	11,301	1,666	67	180	188

**Table 3.** Performance of eCP index creation policies, two drive setups.

fewer but much larger files (109 files of 128MB). The performance of **TF** differs much from **CF** with magnetic storage devices as many arm movements are done. Interestingly, for **TF**, the SuperTalent SSD performs poorly—unfortunately, not all SSDs are equal, as reported in [4]. In contrast, the Intel SSD completely outperforms all the other setups, showing that it handles random reads and writes very well.

Turning to the assignment phase, Table 2 shows that **CF** spends very little time waiting for I/O. With **CF** we observed much overlapping between CPU computations and disk requests thanks to OS and device optimizations which keep the processor (usefully) busy while waiting for I/O completion. This explains the very small times for **CF**, in particular with the Intel SSD which proves to handle competing reads and writes very well. With **TF**, the assignment phase is CPU bound, suggesting a look at parallelism.

The merging phase for **TF** is costly due to the multitude of (relatively) small file accesses compared to **CF**. Merging for **CF** also greatly benefits from the prefetching done by the OS: the few large files are brought into memory before the data gets processed, reducing I/O cost. Prefetching is less profitable for merging with **TF** as many small files are involved.

### 5.3 Two Drive Experiment

By using separate physical drives for the reading and writing, competition for the disk is potentially eliminated. We observed that the larger costs occur when writing the assigned vectors to disk and then reading them back as in both cases many random accesses are performed; using an SSD is therefore ideal to speed up indexing. We defined two setups: First, we kept the input and final output on the Fujitsu (the magnetic disk with the best observed performance) but used an Intel SSD for the intermediate files. The second setup used two identical Intel SSDs. The performance measurements for these setups are reported in Table 3.

The table shows that using the SSD for costly random operations provides dramatic total time improvements, regardless of the type of the other device. With the Fujitsu-Intel-Fujitsu setup, the Intel 66MB/s write speed matches closely the Fujitsu 68MB/s read capacity. Replacing this magnetic device with an SSD does not help much as their total times are very similar. One key lesson is that it is not necessary to put SSDs everywhere, which could be terribly expensive, but to use them solely where random accesses are massively needed. This greatly reduces costs, both in terms of performance and money. Note that it is the CPU cost that dominates the time for **CF**, with I/Os being relatively cheap. The **TF** policy, however, suffers again from the multitude of small files.

Turning to assignment, CF again outperforms TF since it is dealing with a multitude of small files with blocking write accesses. CF with Fujitsu-Intel-Fujitsu is limited by the time it takes to read the data from the magnetic source: it has a lot of CPU to do once the *in-buff* buffer gets filled and the disk is not accessed again for some time, long enough to have the disk entering a power-saving mode, typically reducing its rotational speed. This, in turn, increases the cost of the next data request. The Intel-Intel'-Intel setup has no such problems and its performance is extremely good. It turns out to be slightly above what was observed in Table 2; the reasons are unclear, but some fluctuations have been observed. Note, however, that 67 seconds are insignificant with respect to the total time of more than 13 thousand seconds.

The quite small values during merging, for both policies and both setups, show the improvements from the lack of competition between reads and writes as they are directed to different drives. Even SSDs suffer from I/O competition.

#### 5.4 Other Results

We also checked the impact of a larger *in-buff* on the performance, both for the single drive and two drive setups. As expected, enlarging *in-buff* speeds up TF as large *in-buff* reduces the number of random writes that are necessary. In contrast, larger writes slow down CF because the OS is better at overlapping CPU and I/Os when performing writes in bursts as large writes overwhelm buffers.

SSD performance degradation over time has been reported in other studies, especially for certain IO patterns (e.g., see [2, 3]). We therefore monitored the performance of our SSDs over time but did not find any significant performance change. Our pattern of always ending with a large sequential write may work to the advantage of the SSDs by preventing such degradation.

## 6 Conclusion

We have created two very different implementations of a rather simple, yet very effective, cluster-based high-dimensional indexing strategy. These implementations differ in the way they access secondary storage, emphasizing small vs. large I/Os or random vs. sequential I/Os. We have then run these implementations on a machine connected to various magnetic storage devices, as well as various SSDs devices, and measured the performance.

Our results show that the secondary storage devices used for large scale high-dimensional indexing are key to performance. On the one hand, a carefully crafted implementation can get good performance when using traditional magnetic devices. On the other hand, simpler implementations, potentially saving RAM, can perform very well when high performance SSDs devices are used, as they cope very well with random accesses. SSDs, however, are not the magical answer to all performance problems: their capacity is still limited; their price is so far very high, although this will probably quickly change; and their observed performance varies tremendously from one model to the other.

But generalizing this result, one should carefully evaluate the cost of an extremely sophisticated implementation versus buying efficient storage devices and placing them along the performance-critical paths.

*This work was partly achieved as part of the Quaero Project, funded by OSEO, French State agency for innovation.*

## References

1. A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM*, 51:117–122, Jan. 2008.
2. M. Athanassoulis, A. Ailamaki, S. Chen, P. B. Gibbons, and R. Stoica. Flash in a dbms: Where and how? *IEEE Data Eng. Bull.*, 33(4):28–34, 2010.
3. P. Bonnet and L. Bouganim. Flash device support for database management. In *CIDR*, pages 1–8. [www.crdrrdb.org](http://www.crdrrdb.org), 2011.
4. L. Bouganim, B. T. Jónsson, and P. Bonnet. uFLIP: Understanding flash IO patterns. In *Proc. CIDR*, 2009.
5. M. Casey, R. Veltkamp, M. Goto, M. Leman, C. Rhodes, and M. Slaney. Content-based music information retrieval: Current directions and future challenges. *Proceedings of the IEEE*, 96(4):668–696, april 2008.
6. F. Chierichetti, A. Panconesi, P. Raghavan, M. Sozio, A. Tiberi, and E. Upfal. Finding near neighbors through cluster pruning. In *Proc. PODS*, 2007.
7. R. Datta, D. Joshi, J. Li, and J. Z. Wang. Image retrieval: Ideas, influences, and trends of the new age. *ACM Comput. Surv.*, 40:5:1–5:60, May 2008.
8. G. Gudmundsson, B. T. Jónsson, and L. Amsaleg. A large-scale performance study of cluster-based high-dimensional indexing. In *Proc. ACM-MM-Workshop on Very-Large-Scale Multimedia Corpus, Mining and Retrieval*, 2010.
9. H. Jégou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *IEEE TPAMI*, 33(1):117–128, 2011. to appear.
10. H. Lejsek, F. H. Ásmundsson, B. T. Jónsson, and L. Amsaleg. NV-Tree: An efficient disk-based index for approximate search in very large high-dimensional collections. *IEEE Trans. Pattern Anal. Mach. Intell.*, 31:869–883, May 2009.
11. M. S. Lew, N. Sebe, C. Djeraba, and R. Jain. Content-based multimedia information retrieval: State of the art and challenges. *ACM Trans. Multimedia Comput. Commun. Appl.*, 2:1–19, February 2006.
12. D. G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2), 2004.
13. L. Paulevé, H. Jégou, and L. Amsaleg. Locality sensitive hashing: A comparison of hash function types and querying mechanisms. *Pattern Recognition Letters*, 31(11):1348 – 1358, 2010.
14. J. Philbin, O. Chum, M. Isard, J. Sivic, and A. Zisserman. Lost in quantization: Improving particular object retrieval in large scale image databases. In *Proc. CVPR*, 2008.
15. H. Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.
16. U. Shaft and R. Ramakrishnan. Theory of nearest neighbors indexability. *ACM TODS*, 31(3):814–838, 2006.
17. J. Sivic and A. Zisserman. Video Google: A text retrieval approach to object matching in videos. In *Proc. ICCV*, 2003.