Cluster Computing (2024) 27:3499–3519
https://doi.org/10.1007/s10586-023-04159-8

Large-scale response-aware online ANN search in dynamic datasets

Guilherme Andrade[1] · Willian Barreiros Jr.[1] · Leonardo Rocha[2] · Renato Ferreira[1] · George Teodoro[1]

Received: 11 July 2023 / Revised: 24 August 2023 / Accepted: 13 September 2023 / Published online: 14 October 2023

Abstract
Similarity search is a key operation in content-based multimedia retrieval (CBMR) applications. Online CBMR applications, which are the focus of this work, perform a large number of search operations on dynamic datasets, which are updated at run-time. Additionally, the rates of search and data insertion (updated) operations vary during the execution. Such applications that rely on similarity search are required to fulfill these demands while also offering low response times. Thus, it is common for the computing demands in such applications to exceed the processing power of a single computer, motivating the usage of large-scale compute systems. As such, we propose in this work a distributed memory parallelization of similarity search that addresses these challenges. Our solution employs the efficient Inverted File System with Asymmetric Distance Computation algorithm (IVFADC) as the baseline, which is extended to support dynamic datasets. A dynamic resource management algorithm, called Multi-Stream Adaptation (MS-ADAPT) is proposed. It allows run-time changes on resource assignment with the goal of reducing response times. We evaluate our solution with multiple data partitioning strategies using up to 160 compute nodes and a dataset with 344 billion multimedia descriptors. Our experiments demonstrate superlinear scalability and MS-ADAPT outperforms the best static approach (oracle) by improving the response times up to 32× on high-load cases.

Keywords Online multimedia similarity search · Approximate nearest neighbors search · Product quantization ANN · Distributed computing

1 Introduction

Content-based multimedia retrieval (CBMR) is increasingly important due to the growth of its use in several applications [1–5]. Example applications employing CBMR include content-based image search systems, video identification and misuse, and several applications in social networks such as data propagation. A common aspect with these examples is the large and increasing amount of data employed. A CBMR application may consist of multiple processing steps, but at its core is the operation that finds similar objects to an input query object in a reference dataset (index). The multimedia objects are represented using high-dimensional vectors (descriptors) with hundreds to thousands of dimensions [6–8]. Thus, the similarity search here consists of finding the nearest feature vectors in the dataset to the query descriptor(s) measured by a distance metric, corresponding to computing the k-nearest neighbors (k-NN) search problem [9].

The exact solution for the k-NN via exhaustive search is not viable in CBMR because it computes distance between

✉ George Teodoro
george@dcc.ufmg.br

Guilherme Andrade
gandrade@dcc.ufmg.br

Willian Barreiros Jr.
willianjunior@dcc.ufmg.br

Leonardo Rocha
lcrocha@ufsj.edu.br

Renato Ferreira
renato@dcc.ufmg.br

[1] Department of Computer Science, Universidade Federal de Minas Gerais, Antonio Carlos, Belo Horizonte, Minas Gerais 31270, Brazil

[2] Department of Computer Science, Universidade Federal de São João Del Rei, Rodovia BR-494, São João Del Rei, Minas Gerais 36301, Brazil

a query and each high-dimension descriptor in a huge dataset. Several indexing structures, such as kd-tree, k-means trees etc [10–12], have been proposed to reduce the k-NN search cost. These structures organize the data spatially to reduce the distance computations in the search for a dataset subset. While they attained remarkable performance for datasets with small dimensionality, their ability to prune the search degrades with high-dimensional descriptors as used in CBMR [13].

These limitations motivated the development of approximate nearest neighbors (ANN) search algorithms for applications that can trade off accuracy for performance. The ANN approaches were then created to index and search very quickly in large and high-dimensional datasets while maintaining high response accuracy. Popular and efficient ANN indexing solutions include Locality-Sensitive Hashing (LSH) [14] and Inverted File System with Asymmetric Distance Computation (IVFADC) [15]. These algorithms are built on the concept of partitioning the space into buckets and enabling the search to take place in a subset of buckets. In particular, IVFADC has presented superior performance compared to competitors [15] while using a fraction of the memory demanded by them. As such, we use and parallelize IVFADC in this work.

Besides the advances with recent ANN indexing strategies, there are several challenges to be addressed in the context of online multimedia retrieval applications, which are the focus of our work. First, most of the ANN indexing strategies have developed sequential algorithms, but the computing and memory requirements to search on a huge dataset used by target applications exceed the capacity of a single node. Second, several real-world online applications have to deal with dynamic datasets that are updated during the execution, but most solutions can only deal with static datasets. Third, the load of queries (and index updates) submitted to the system varies during the execution in online applications. Therefore, in order to minimize the end-user observed response times, it is necessary to adjust the system parallelism and resource allocation at run-time according to the load observed.

In order to address these challenges, in this work, we propose an efficient distributed memory system for the execution of ANN in large-scale datasets. Our solution deploys the efficient IVFADC algorithm and evaluates multiple data distribution/partition approaches. Further, we have developed novel strategies to handle dynamic datasets. Our solutions are based on the concept of data buckets partitioned in time and space, as opposed to the space only approach used in previous works [16, 17] that could only handle static datasets. This allows to quickly discard unnecessary (outdated) data from the index while also reducing synchronization overheads due to concurrent indexing and searching. Finally, we have also developed a novel algorithm called MS-ADAPT that configures the system according to the observed input load. This run-time optimization targets the goal of minimizing user response times. MS-ADAPT distributes the resources among querying and indexing tasks according to their demands, which also change during the execution. The contributions of this work may be summarized as follows.

- We develop a distributed memory system for execution of ANN search using IVFADC. Our solution includes a new data organization to support dynamic datasets. In this setting, the data temporal aspect is considered when instantiating ANNs algorithms' data partitions (buckets). This process enables search and insertion to occur concurrently in different buckets or their temporal partitions;
- We propose an adaptive resource assignment algorithm called MS-ADAPT tune the computing resource allocation among index update and search. Our approach aims to adapt the system during the execution as the resource allocation varies at run-time according to the load submitted to the system. MS-ADAPT was extensively compared to the best static approach (oracle) and presented the same or superior performance, reducing response times in up to about 32× for cases with high loads;
- The propositions are evaluated at scale using up to 160 computing nodes with a dataset of 344 billion SIFT multimedia descriptors. The results show that our solutions present super-linear scalability and low response times even when huge datasets are used.

2 Problem statement and background

2.1 Problem statement

The problem of k-nearest neighbor (k-NN) search can be outlined as follows. Given a dataset Y containing l vectors $(y_i \in Y | i \in 1 \ldots l)$ in a D-dimensional space (R^D), and a query vector $x \in R^D$, the goal of k-NN search is expressed by:

$$L = \text{k-}\underset{i=1\ldots l}{\operatorname{argmin}}\ dist(x, y_i), \tag{1}$$

Here, L constitutes a list that captures the k closest points to x within Y. The commonly used distance metric in this context is the Euclidean distance. However, computing exact k-NN involves calculating the distance between the query and every vector in Y, which can be computationally expensive for large or high-dimensional datasets. Consequently, we use approximate nearest neighbor (ANN) search, which provides a balance between exactness and

speed. For this purpose, we adopt the efficient IVFADC ANN indexing approach [15].

Our focus lies on applications related to online multimedia retrieval, which demands the indexing of large and dynamic datasets and answer to varying query rates. Parallel distributed memory machines offer a powerful computational environment for such applications, but most of existing ANN parallelization techniques designed for this environment have targeted batch execution for static datasets [12, 18–22]. However, online applications have to deal with dynamic datasets and must quickly answer incoming queries to optimize response times under fluctuating query rates. These applications should optimize their system configuration for distributing compute power between query processing and data update. However, configurations optimized for batch execution throughput often differ from configurations which optimize response times under variable query rates.

We expand on the ongoing research with a parallelization strategy involving distributed memory. We have devised novel strategies for managing dynamic datasets. Our solutions use the concept of temporal and spatial partitioning of data buckets. This approach enables the quick elimination of obsolete data and reduces resulting synchronization from indexing and searching processes. We also developed novel strategies to dynamically configure the system based on run-time load measures with the objective of reducing user-observed response times. Our strategy effectively allocates resources to querying and indexing tasks in accordance with their respective demands, which inherently fluctuate during execution.

2.2 Inverted file system with asymmetric distance computation (IVFADC) indexing

This section presents the product quantization concepts, their use in nearest neighbor search and the indexing structure employed to reduce distance computations in large datasets by IVFADC [15].

2.2.1 Product quantization concepts

Quantization involves the reduction of the cardinality of a space's representation. Given a vector x with D dimensions, a quantizer, denoted as q, maps x to a vector $q(x)$ within the set $C = c_i; i \in Z$, where Z is a finite index set of size $0...k-1$. The values c_i are referred to as centroids, constituting the codebook C. In the k-means algorithm that is typically employed to build C from a subset of the dataset, every cluster (data points grouping) is symbolized by its center, termed a "centroid", signifying the arithmetic mean of the data points allocated to that cluster. This centroid acts as a representative data point that embodies the cluster's central location. Effectively quantizing high-dimensional vectors necessitates a large codebook to minimize quantization error [15]. However, the impracticality here arises from the associated high quantization cost and the substantial memory requirement for storing C when it is very large.

To address these challenges, the concept of product quantization has been proposed. Using quantization, the vector x is partitioned into m subvectors u_j ($j \in 1...m$) each with $D^* = D/m$ dimensions, assuming D is a multiple of m. Quantization of x is formally defined as follow:

$$\underbrace{x_1,\ldots,x_{D^*}}_{u_1(x)},\ldots,\underbrace{x_{D-D^*+1},\ldots,x_D}_{u_m(x)} \rightarrow q_1(u_1(x)),\ldots,q_m(u_m(x)) \tag{2}$$

This leads to the quantization of x through a Cartesian product of its quantized subvectors:

$$q(x) = q_1(u_1) \times q_2(u_2) \times ... \times q_m(u_m) \tag{3}$$

The quantized vectors are expressed as compositions of subvector quantization indices ($\tau = \tau_1 \times \ldots \times \tau_m$), resulting in a codebook $C = C_1 \times \ldots \times C_m$ with C_j representing centroids used for individual subvector quantization. Assuming each subvector employs k centroids, the codebook C encompasses k^m quantization combinations. Thus, using these low-complexity quantizers in subvectors enables us creating a very large number of quantization combinations.

To proceed with the utilization of quantization in nearest neighbor search, the computation of distances within quantized spaces is crucial. Among approaches proposed in [15], the Asymmetric Distance Computation (ADC) proves to be the most efficient. ADC calculates the distance between a query object x and a quantized database document y as follows:

$$\tilde{d}(x,y) = d(x,q(y)) = \sqrt{\sum_{j=1}^{m} d(u_j(x), q_j(u_j(y)))^2} \tag{4}$$

2.2.2 Searching in quantized spaces

Even within a lower-dimensional space, performing a thorough exhaustive search is not feasible. As a solution, the application of an inverted file system with Asymmetric Distance Computation (IVFADC) was suggested to streamline the search process. This method involves storing data elements that have been quantized to the same centroid into the same inverted file system entry. Consequently, the search is constrained to a limited number of inverted file entries linked to centroids that are closest to the query.

Algorithm 1: Indexing and Searching using IVFADC

```
1:  function Indexing(y)
2:      k' ← q_c(y) ;
3:      r_y ← y - C_c[k'];
4:      code ← q_p(r_y);
5:      ivf[k'].append(y.id, code);
6:  function Search(x, w, k)
7:      nnc ← kNN(C_c, x, w);
8:      for i ∈ 1 ... w do
9:          r_x ← x - C_c[ncc[i]];
10:         distList ← dist(index[ncc[i]], r_x);
11:         ann ← k-select(ann, distList,k);
12:     return ann;
```

Figure 1 shows the indexing structure using an inverted file system, where each entry is associated to a representative centroid descriptor computed with a clustering process using k-means on a training dataset. Each inverted index entry contains a list of quantized objects (id and code). The *id* can be an application specific identifier of the original object that was quantized. The *code* is information used to improve the approximation of the ADC calculation, being the distance between the stored descriptor and the corresponding closest centroid. Additionally, *code* is also quantized with Product Quantization to reduce storage demands.

The IVFADC indexing and searching phases are presented in Algorithm 1. Indexing follows these steps:

1. Calculates the closest centroid of the input descriptor y (line 2).
2. Calculates the residual value r_y from y to k′-th (closest) centroid ($C_c[k']$) (line 3).
3. Calculates *code* through Product Quantization of residual value (line 4).
4. Stores *id* and *code* on an entry of the inverted file system associated with the nearest centroid (line 5).

Indexing phase produces the inverted file system where database descriptors are quantized and stored in corresponding inverted lists (buckets). Then, the search phase takes as input: (i) the query descriptor (x), (ii) the amount of inverted index entries to visit (w); (iii) and the number of nearest neighbors to retrieve (k). Search phase will perform the following steps:

1. Calculates the closest w centroids and store them in *ncc*
2. Searches in the inverted index entries associated with each of these centroids (lines 8 to 11). For each entry:
   (a) Calculates the residual of x to the respective centroid (line 9).
   (b) Calculates the ADC distances for all descriptors stored in the inverted index entry (line 10).
   (c) Selects the closest k descriptors and add them to the set of found ones (line 11).

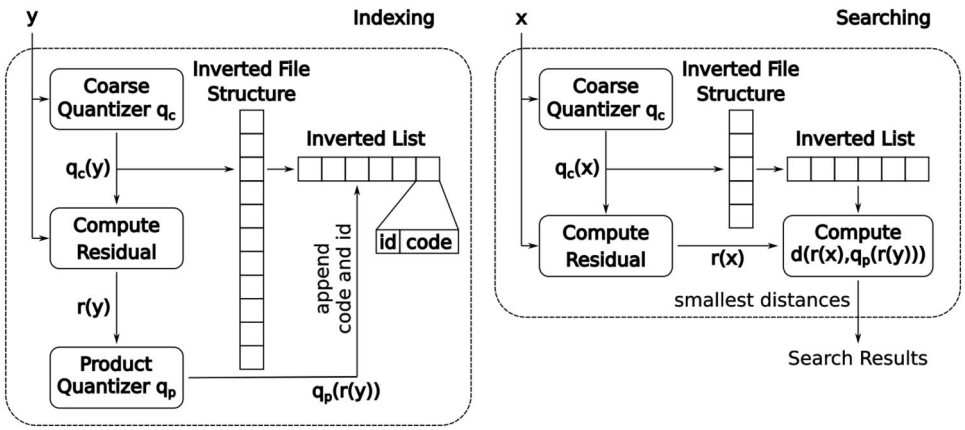


Fig. 1 ANN using inverted file system and ADC (IVFADC)

3. After *w* entries are visited, the result of the *ann* set is returned.

3 Related work

Most seminal works on ANN algorithms have focused on proposing and improving indexing methods targeting sequential single node machines [14, 15, 22–25]. However, as previously discussed, a single node's computing and memory capabilities are typically insufficient to execute modern CBMR applications efficiently. Therefore, in recent years, the development of distributed memory and GPU accelerated ANN algorithms have attracted the attention of the industry and academic communities, which resulted in significant advances in the domain [12, 18–20, 26–31]. The rest of this section focuses on discussing the parallel and distributed memory ANN systems.

The distributed ANN parallelization proposed by [18] employed MapReduce [32]. This strategy relied on the file system to maintain indexed data, which were retrieved to the main memory during query time. Consequently, it focused on throughput/batch-oriented searching due to the high query response time. This strategy was optimized with an in-memory Active Distributed Hash Table [19] to store the index. Both solutions have built parallel versions of LSH. [20] proposed a distributed memory Index Tree similarity search implemented using MapReduce with the file system storing the index. This work innovated in the use of query reordering to reduce data movements. The paper [30] extended LSH by using sketches to prune distance computations and implemented a parallel version using the Message Passing Interface (MPI) [33].

The Faiss library [31] developed by the Facebook AI research group is another important tool in the domain. It includes very efficient implementation of multiple algorithms targeting CPU and GPU. Here, we leverage their efficient code to implement the IVFADC in each local node of our multi-node implementations and extend it to support temporal indexes. Another popular work [34] used Spark [35] with its Resilient Distributed Datasets (RDD) structure to implement an ANN algorithm on the distributed memory system. While it proposes and evaluates the system in the context of a real-world setting, this solution still suffers from high response times. This occurs because the system is optimized for large query batches. The work presented in [36] proposed a parallelization of the Hierarchical Navigable Small World Networks (HNSW) ANN algorithm [22]. They proposed a Spark-based parallelization where the dataset is divided among nodes using a Data Equal Split (DES) strategy, and each dataset partition is indexed locally by a modified HNSW algorithm version that partitions the data creating a two-level approach. While efficient, HNSW suffers from high memory use and their experiments were limited to 180 M data points.

Only a few works in the literature consider the online scenario in which (i) the dataset is dynamic or updated at run-time and (ii) the load (query rates) submitted to the system varies during the execution. The work of [29] is a seminal solution to address dynamic datasets, and is considered to be the state-of-the-art. Their strategy consists of a distributed memory implementation of LSH designed to handle the insertion of new data during execution. It has been implemented using a sliding window scheme in which a subset of the compute nodes handle insertion with a data structure called *Delta Table*. This structure is then periodically merged into the static part of the index. Unfortunately, this system does not handle varying query loads and is restricted to a specific parallelization strategy, while here, we developed multiple parallelization strategies that also support dynamic settings. Because this is an important reference for indexing and querying in dynamic datasets, we used its parallelization strategy as a baseline for comparison in the experimental evaluation.

The work of [37] has also addressed dynamic datasets by supporting indexing updates. They built a shared-memory parallel version of LSH that takes advantage of RAM and flash memory together to reduce the impact of updates on the system performance. Another solution that deals with dynamic datasets to support streaming data ANN is presented in [38]. The authors leverage a graph-based algorithm and enable quick updates using SSD combined with RAM. In this case, updates are computed in memory, while other parts of the indexed data are stored in the SSD. Unfortunately, this work is also limited to a single node.

In other recent works, we have developed strategies to deal with varying query loads [21, 27]. First, we parallelized Hypercurves [27] to allocate the processing in CPU and GPU under varying loads to minimize response times. We have further developed a distributed memory version of IVFADC [21] targeting GPU. The latter work tunes at run-time the number of queries bundled for execution with the GPU targeting response time reduction. In both cases, however, the dataset is considered static, and only a single data partition strategy is used in the distributed memory execution. We have also developed a parallel framework for executing ANN search algorithm on distributed machines in which multiple data partitions strategies were proposed [17]. This system is the baseline work in which our contributions are built and the data partitions it implemented are presented in the next section.

Table 1 categorizes the related work on distributed ANN according to the main features required for a fast

Table 1 Comparison between distributed ANN search strategies

Indexing algorithm	In-memory	Data partition	Locality-aware	Dynamic datasets	Varying query rate
LSH [18]	✗	BES	✗	✗	✗
LSH [19]	✔	BES	✗	✗	✗
Index Tree [20]	✗	BES	✗	✗	✗
eCP [34]	✔	BES	✗	✗	✗
LSH [29]	✔	DES	✗	✗	✗
FLANN [12]	✔	DES	✗	✗	✗
Multicurves [27]	✔	DES	✗	✗	✔
IVFADC [21]	✔	DES	✗	✗	✔
SLASH [30]	✔	DES	✗	✗	✗
HNSW [22]	✗	DES	✗	✗	✗
LC [39, 40]	✔	BES	✗	✔	✗
LSH [29]	✔	DES	✗	✔	✗
IVFADC [17]	✔	DES/BES SABES SABES++	✔	✗	✗
This work (IVFADC)	✔	DES/BES SABES SABES++	✔	✔	✔

online search of dynamic datasets: (1) in-memory indicates whether the index is stored in fast memory (RAM); (2) data partition employed for distributed execution: DES that divides data equally among nodes, BES that divides buckets of data among nodes and SABES/SABES++ that considers data spatial organization in the partitioning. These strategies are discussed in Sect. 4.3; (3) if data partition is locality-aware/organizes the index targeting to minimize the number of nodes involved in a search; (4) if dynamic datasets (run-time index update) are supported; and, (5) indicates if the solution is optimized to deal with query load variations expected to occur in the online system due to the user interaction. As may be noticed, our work is the only one to support multiple data partition strategies and includes data-aware partitioning that significantly improves performance compared to other approaches. Also, when it refers to dynamic datasets and varying query loads (rates), our work is the only solution to address both aspects, which are essential to improve performance and user experience in modern CBMR applications.

4 Scalable distributed memory parallelization

This section presents the distributed memory architecture proposed to address the challenges of ANN in large data volumes. This solution intends to allow for rapid deployment on distributed environments enabling the use of different data partitioning strategies. This architecture extends the approach proposed in [16] with the ability to handle dynamic datasets. The next sections detail the architecture and the proposed parallelization strategies.

4.1 Architecture

The system architecture is built using a set of processes organized into a dataflow scheme. Each dataflow stage may be instantiated multiple times depending on its computation demands. The stages used in our solution are: (i) Readers/Object Streamers: responsible for distributing the initial dataset and streaming of data objects (updates) received during the execution among Query Processors (QP); (ii) Coordinators (Co): that receive the query stream and send queries to a specific set of QPs and aggregate the global set of nearest neighbors retrieved by those QPs; (iii) Query Streamers (QS): responsible for generating queries stream; (iv) Query processors (QP): responsible for maintaining a local index in each node in which it is instantiated and for computing the ANN local search. The distributed memory solution organizes the stages into the Index building and Search/Production phases.

The *Index Building Phase* works with two steps. The first step is responsible for configuring the IVFADC indexing data structure, which consists of computing a set of *Representing Centroids* (C_c). In the second step, through the method *sendTo*, data objects are distributed among

QPs following some partitioning strategy. Objects received by QPs are then kept locally after indexing.

The *Search/Production Phase* handles the execution of object insertions (updates) and query streams. The Query Streamers (QS) processes send queries to the system, which the Coordinators (Co) receive (Fig. 2). The query message is forwarded, following the implementation of the *forward* method defined by the underlying partitioning strategy to the Query Processors (QP) nodes responsible for processing it. The Coordinator waits until the local searches are finished and the QPs return local results to be aggregated into a global response. In parallel and independently, the Object Streamer (OS) processes distribute new descriptors among QPs (bottom part of Fig. 2). In essence, they generate a stream of data being received to be indexed during the execution. These data are processed using the same strategy of the index building in which the *sendTo* method computes the QPs to which descriptors are sent according to the partitioning strategy implemented. Details on how the QPs efficiently handle on-the-fly indexing are presented in Sect. 5.

4.2 Intra-stage parallelization

We have developed the QPs as a multithreaded stage using the Pthreads library. Intra-node parallelization is necessary due to this being the most compute-demanding stage as it executes the actual local nearest neighbors. With the intra-node shared memory parallelization, we only need a single QP process running per compute node, which reduces the number of data partitions to one per node, also reducing overall interprocess communication.

As shown in Fig. 3, the QP internal architecture is implemented using the following threads: (i) Reception Thread: responsible for receiving queries and descriptors. The queries are stored into a query queue (QQ) for further processing, while new objects are stored into the object queue (OQ). (ii) Local Search Threads: are workers threads that, when idle, access the QQ and retrieve a new query, seen as (1.c) in Fig. 3. This type of worker will perform a local search process (*search* method of the Temporal Index discussed in Sect. 5), visiting *buckets* of interest and finding the nearest *k*-neighbors locally according to the ANN algorithm instantiated in the system on (1.d). When the local search finishes, the results are sent to the Coordinator responsible for that particular query; (iii) Insertion Threads: are in charge of retrieving objects from the OQ (2.c), and inserting them into the index (2.d). The insertion of a new object into the index is a very fast process. As insertion and searching takes place in parallel, adequate control access is implemented to avoid race conditions, discussed in Sect. 5. (iv) Stream Controller Thread: in our

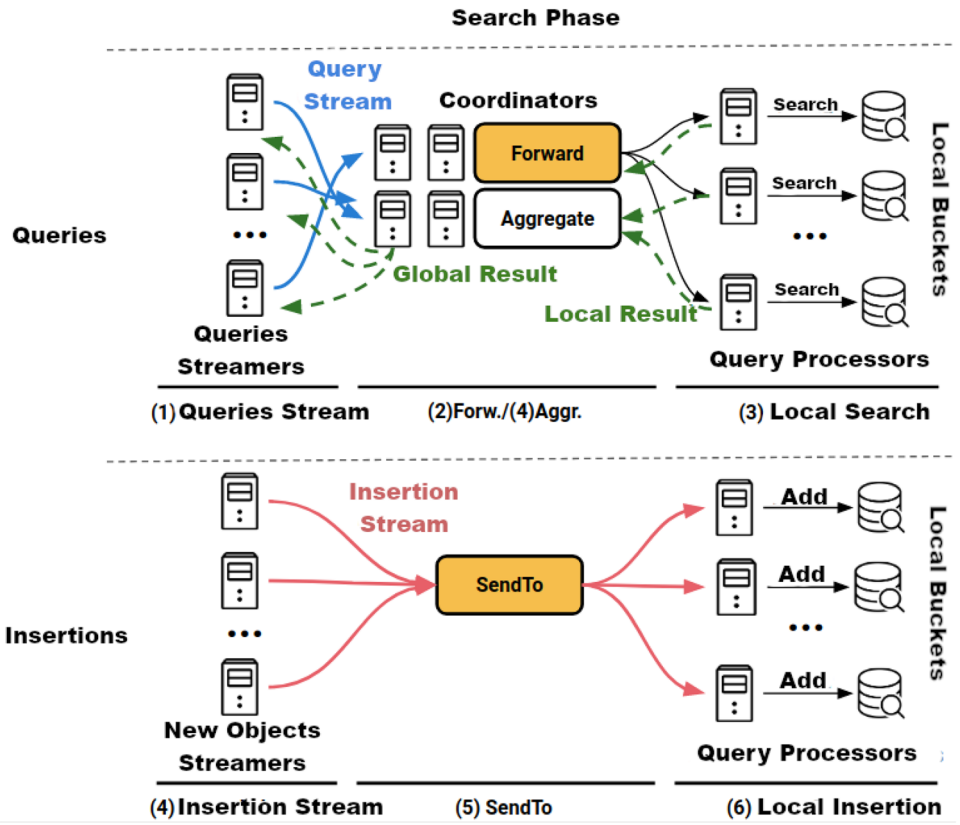


Fig. 2 The search phase processes queries (top) and executes descriptors insertions (bottom) in parallel. The searches involve the QS that sends queries to Cos, which, in turn, forward them (using the *forward* function) to the QPs responsible for the search. The QPs retrieve their local nearest objects and send them back to the Co for aggregation, then generating the final results. The insertion involves the Object Streamers that send new objects to the QPs according to the data partition strategy implemented in the *sendTo*

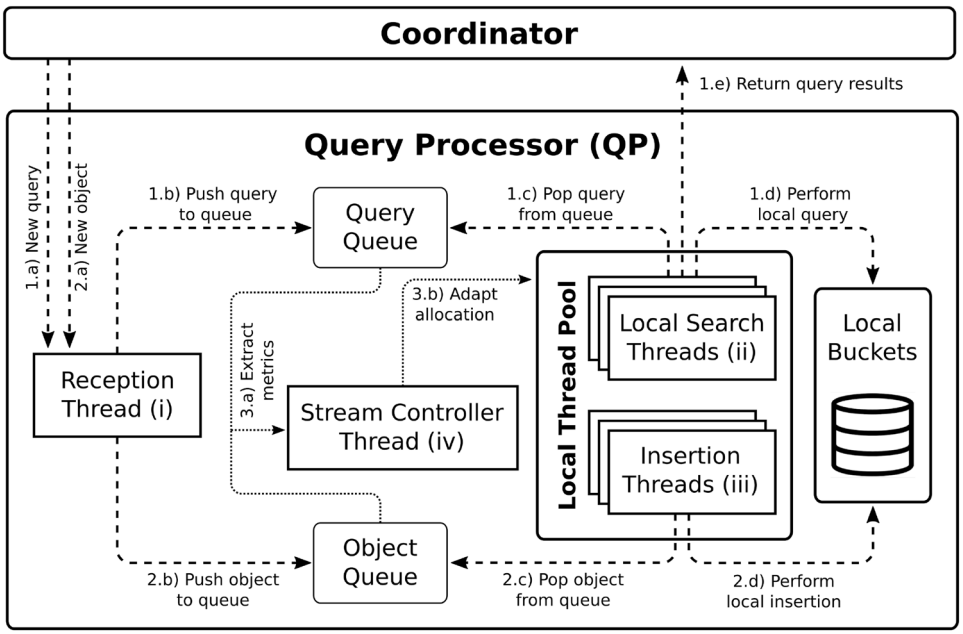


Fig. 3 Internal organization of a Query Processor (QP). It interacts with the Coordinator, receiving queries and objects to insert, and returns query results. The internal thread pool allocation adapts to the current state of both query and object queues. The figure also exemplifies the query (1) and insertion (2) workflows

solution that implements runtime tuning of the resource allocated to handle input data insertion and query processing, the Stream Controller thread is in charge of monitoring the system (3.a) and changing the assignment of threads to insertion and query processing (3.b). The design and implementation of the adaptation method executed by the Streams Controller are presented in Sect. 6. This is executed with the goal of reducing the response times observed by the users.

4.3 Data partitioning strategies

This section describes the data partition strategies implemented and evaluated in our system. This includes traditional approaches found in the literature (DES and BES), and the Spatial-Aware Bucket Equal Split (SABES) and Spatial-Aware Bucket Equal Split with Data Balancing (SABES++) proposed in our work. The development of strategies is performed by only changing the *forward* and *sendTo* routines of the system.

4.3.1 Data equal split (DES)

The DES strategy splits the dataset equally among the QP processes in a *round-robin* fashion during the construction phase. This means that for each centroid, each entry within it is distributed evenly, one at a time, to each QP, being all centroids represented on all QPs, as shown in Fig. 4. This can be done by using the *sendTo* routine. Consequently, in each QP there will be entries in the local index for all buckets, while the descriptors are distributed among QPs. Thus, the search requires visiting all QP nodes in the distributed memory machine and the *forward* method implements a query *broadcast*. This strategy was found in several previous works [12, 28, 29].

Using the IVFADC, which is the basis of this work, each QP has a complete IVF structure with entries associated with all representative centroids (Cc). By sending the query to all QPs, it is guaranteed that the w lists associated with the closest centroids will be fully visited, since the descriptors in these lists are divided among QPs. Figure 4 illustrates the described process.

4.3.2 Bucket equal split (BES)

The *Bucket Equal Split* strategy is based on the distribution of buckets among QPs. Thus, using IVFADC, the inverted file entries are evenly distributed among the QPs and this distribution ensures that all objects in a given bucket will be stored together in a single QP. Since $|C_c|$ centroids are available, the ith QP (QP_i, $i \in 1...n$) stores $|C_c|/n$ centroids. Custom implementations of *sendTo* and *forward* are presented in Algorithm 2. Figure 4 illustrates the data organization proposed by BES.

During the bucket distribution phase, *sendTo* method receives an object y to be indexed and calculates its closest centroid k', which in turn, is associated with the inverted list entry in which y should be stored (line 2). The mapping from the centroid to the QP instance is then computed (line 3) and returned (line 4).

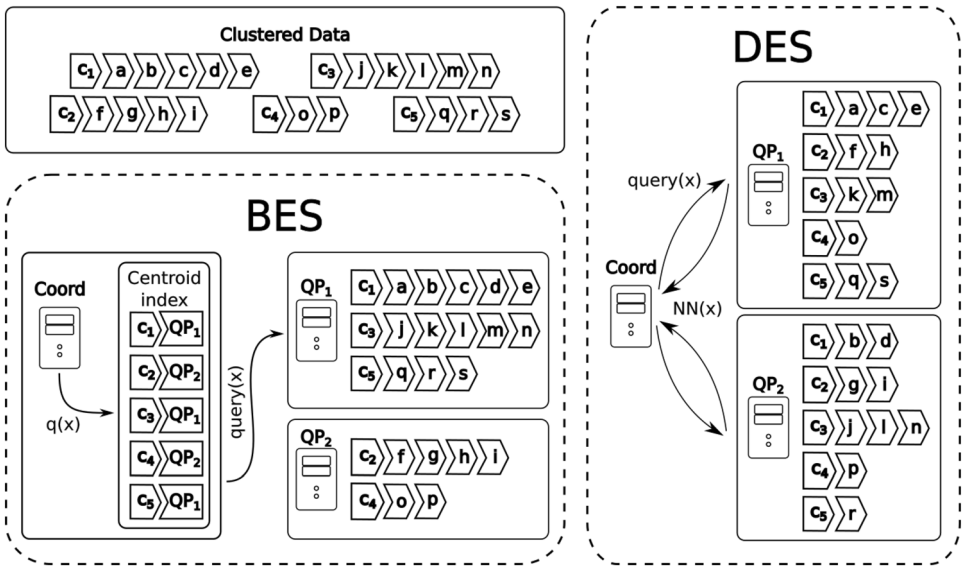


Fig. 4 DES and BES distribution approaches using IVFADC. For this example, initial data points $a - s$ are organized in centroids $c_1 - c_5$. In DES, all QPs have IVF entries for all centroids, and the associated lists have descriptors divided among several different machines. In BES, centroids are distributed among QPs with all its descriptors. For BES, the coordinator has a mapping of centroids to QPs

Algorithm 2: SendTo and Forward for BES

```
1: function sendTo(y)
2:     k' ← q_c(y) ;
3:     target ← k'/n ;
4:     return target
5: function forward(C_c, x, w, n)
6:     nnc ← kNN(C_c, x, w) ;
7:     for i ∈ 1 ... w do
8:         target ← target ∪ ncc[i]/n ;
9:     return target;
```

In the search phase, instead of sending the query x to all QPs as in DES, it will use only the set of QPs storing the w buckets represented by the closest centroids of the query x. So, as shown in Algorithm 2, the *forward* method calculates the w closest centroids of x query (line 6) and then finds which are the QP instances storing those centroids (or inverted lists associated with them) (line 8). The BES approach tends to use a smaller number of QPs for searching than DES, as it sends the query object to at most $min(w, n)$, where n is the number of QP instances and w the number of centroids visited. w tends to be much smaller than n in large distributed systems.

4.3.3 Spatial-aware bucket equal split (SABES)

SABES extends BES by proposing a distribution where buckets close in space are assigned to the same QP. The SABES premise is that spatially close buckets have a high probability of being visited by the same query. Consequently, storing sets of buckets close together on the same QP would reduce the number of QPs used to answer queries.

Algorithm 3: Building SABES regions, SendTo and Forward methods

```
1: function regionConstruction(C_c, n)
2:     R ← k-means(C_c, n) ;
3:     return R;
4: function sendTo(R, y)
5:     k' ← q_c(y) ;
6:     target ← getRegion(R, k') ;
7:     return target;
8: function forward(C_c, R, x, w)
9:     nnc ← kNN(C_c, x, w) ;
10:    for i ∈ 1 ... w do
11:        target ← target ∪ getRegion(R, nnc[i]) ;
12:    return target;
```

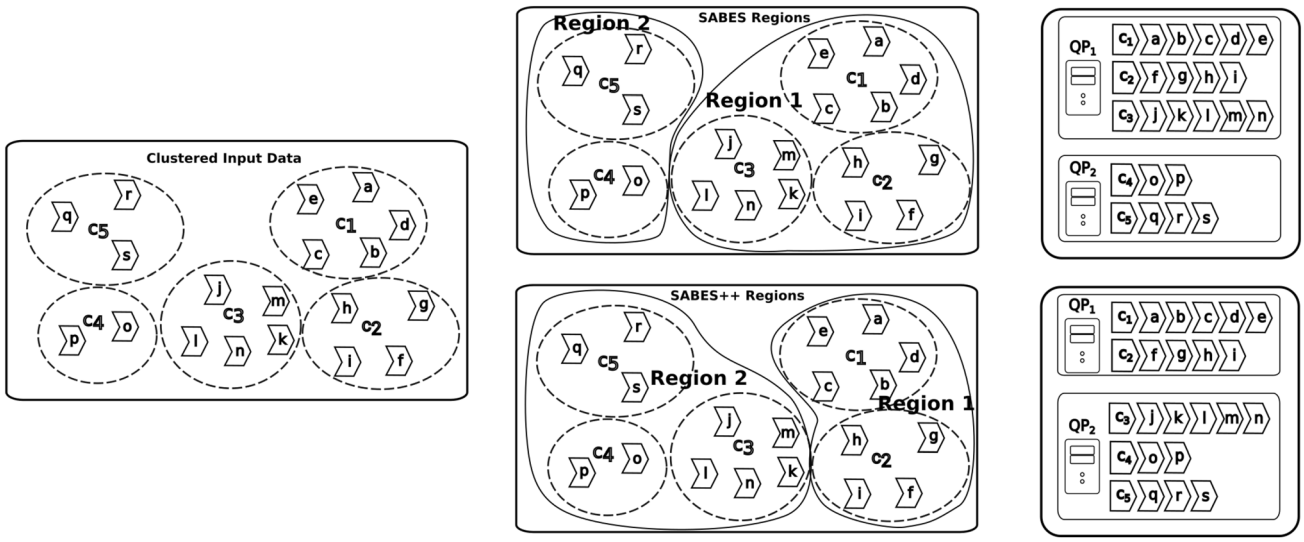


Fig. 5 Data partitioning strategies SABES and SABES++. By considering the number of points in each region, SABES++ reduces workload imbalance for searching

SABES has a pre-processing step where the buckets are grouped into n macro regions R, each macro region R_i will be assigned to a QP_i. Using the IVFADC approach, these regions are calculated using the *k-means* algorithm taking the set of representative centroids C_c as input. This implementation is seen in the *regionConstruction*(C_c, n) method of Algorithm 3. The return R is a dictionary that associates each centroid with the macro region it was grouped into. This dictionary will later be used in the *sendTo* and *forward* methods. The operations *sendTo* and *forward* are also presented in Algorithm 3 and have similar implementations to the same methods present in the BES approach. The main change is the function *getRegion*, which queries the dictionary where centroids are mapped to the macro regions R_i. This function simply accesses the dictionary indexed by the centroid id and returns its respective region. Figure 5 complements the algorithm discussed by showing a visualization of the distribution proposed by SABES.

4.3.4 Spatial-aware bucket equal split with data balancing (SABES++)

SABES++ groups buckets considering their spatial location and the number of objects associated with them. This grouping balances the sum of objects in macro regions and is implemented through the weighted *k-means* algorithm [41] which aims to balance the weights (number of descriptors in each buckets), while at the same time preserving spatial proximity between those in the same group.

Algorithm 4 presents the *regionConstruction* method implemented by the SABES++ for the IVFADC strategy. This method runs offline in a pre-processing phase, and consists of executing the weighted *k-means* (line 2) that calculates the set of centroids (macro regions R). This strategy maintains the spatial proximity between the centroids (*buckets*) allocated in each QP, however, it balances, by best effort, the amount of objects stored in each of the instances. The *sendTo* and *forward* methods for the SABES++ solution are exactly the same as those used by SABES. Figure 5 shows the components of this discussed approach. As may be noticed, it is expected that the number

Algorithm 4: Building SABES++ regions

```
1: function regionConstruction(C_c, n)
2:     R ← weighted-k-means(C_c, n) ;
3:     return R;
```

of points in regions created by SABES++ will be more balanced.

5 Searching in dynamic temporal datasets

The temporal index extension aims to enable efficient searching and indexing of streaming of data. It includes challenges in terms of the data organization and partitioning as it must: (i) accommodate parallel search and insertion of new data and (ii) enable searching in a time window defined by the user, while efficiently releasing old data no longer of interest.

The first aspect defined here is the data stream window model used to partition the data in time. There are two main windowing models: fixed and sliding window [42]. The fixed window partitions the time-space into non-overlapping segments, whereas the sliding window slides in time with a specified time interval (see Fig. 6). While the latter may enable a finer-grained partitioning/searching, the sliding window is harder to be efficiently implemented because search and data removal would have to consider each data element timestamp instead of a fixed range that can be used to group several items. As such, we adopt that fixed windowing model here. The main impact of this decision is that search is carried out in time windows with any time overlap with user defined time window.

The proposed temporal index organizes buckets into two levels: (i) Spatial: the object is stored in the nearest *bucket* as defined by the search algorithm (e.g., for IVFADC) and; (ii) Temporal: an object x in a *bucket* is stored in different partitions (according to the time window model) depending on when object x was received/inserted ($t(x)$). In essence, there is a subpartition of buckets in time. The temporal size of windows (s) in seconds and the number of windows to be searched in the past are application parameters [29]. Figure 7 shows the organization of a local index through *temporal buckets*. Most of the computation in the search will be carried out without interference from the insertion, since only the latest time window bucket partition will receive updates. As data are organized in temporal partitions, the definition of which data should be considered during the search is simple as it is to discard old partitions that are no long required.

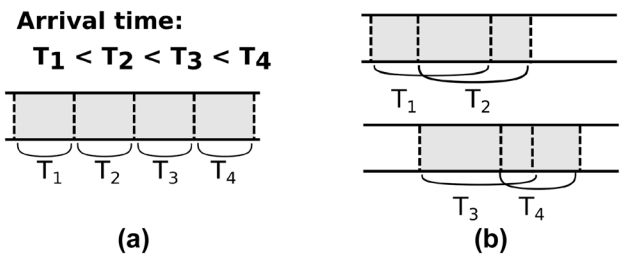


Fig. 6 Windowing models: **a** fixed; **b** sliding

The local insertion and search in the temporal index are performed using the regular algorithms of the ANN method (IVFADC) but with appropriate index partitions. For the insertion, it is always considered the newest partition in time for the respective bucket, whereas the search will consider the time partitions T according to the user input parameters. Our system parallelizes the search internally in each QP process by employing multiple computing threads. In this case, the search is executed in parallel by having each thread independently processing subpartitions of the index. Those subpartitions consist of the w centroids of interest used in IVFADC (spatial partitions) and the temporal partitions within each of those centroids.

It is important to highlight that the operations of indexing new objects and the search process happen simultaneously in the Temporal Index. However, there is concurrency between queries (reading) and index updates (writing) only on the latest temporal window T_n. To manage these concurrent operations, a synchronization based on the *read/write locks* [43] is proposed. This approach ensures that the access to a *bucket* is limited to one indexing operation at a time, but multiple searches can take place concurrently. In this way, before starting a writing operation/insertion in a bucket, the *thread* responsible for the operation obtains the corresponding lock. If successful, the operation is executed. Otherwise, it waits until the *bucket* is freed. This strategy allows the entire Temporal Index to be consulted and new descriptors to be indexed with the cost of synchronizing only one *temporal bucket* partition (the most recent).

6 Run-time resource adaptation

The online scenario to which CBMR applications are submitted is dynamic not only in terms of the dataset being updated at run-time, but also with respect to the query and data insertion rates (load) that vary during the execution. Minimizing the response times in this setting is complex, because resource allocation to handle incoming queries and data insertions that leads to the best response time varies at run-time. Thus, an online dynamic resource allocation strategy is fundamental to achieve a good performance.

The query response time (q_r) may be decomposed into processing time (p) and queue waiting time (wt): $q_r = p + wt$. It is important to highlight that the queue waiting time (wt) is affected by (i) the number of queries waiting to be processed in the query queue (QQ) and (ii) query blocking times that take place when the index is outdated. In this case, if there are object insertions pending or index updates that affect the queries waiting to be

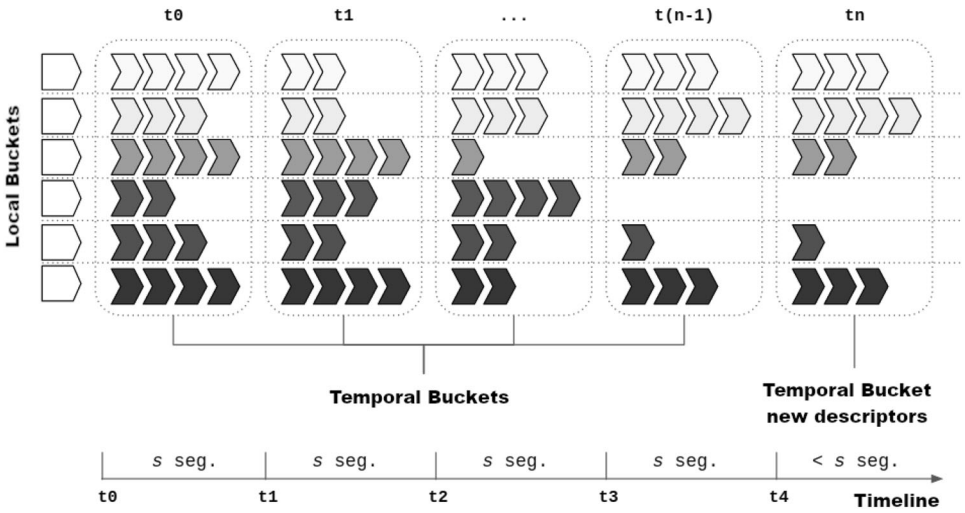


Fig. 7 The temporal index organizes objects by spatial and temporal proximity. Every *s* seconds interval, a new *temporal bucket*/window T is created, and an object *x* with time *t*(*x*) into the appropriate spatial and temporal bucket

processed, those insertions must be processed before the query can be executed. The number of queries waiting and processing blocking times also depend on the computing resources (CPU cores or threads) allocated for query processing (EQP) and insertion (IPP). Thus, the system tuning needs to adjust the values of the tuple [*EQP*, *IPP*] during the execution focusing on reducing the query response time.

The reasoning behind MS-ADAPT is to reduce query processing time while allocating enough resources to insertions so that the index is kept updated over the execution time. Outdated buckets considerably penalize query response time, as queries are blocked until pending insertion tasks are executed. The blocking of query processing will occur if the index outdate is higher than a user defined parameters called Bucket Outdated Flexibility (BOF). In fact, this parameter adds a flexibility to the process that enables insertions to be more appropriately handled, for instance, during periods of lower query processing rates for applications. A similar parameter was used in the previous work [29].

MS-ADAPT presented in Algorithm 5 involves three main steps: (i) Monitoring; (ii) Optimization; (iii) Reconfiguration. The first step collects metrics to analyze the system load: Query Arrival Rate (QAR), Query queue size (QQS), and Maximum outdated bucket (MOB) (line 3). Because MS-ADAPT is executed in define time intervals, these metrics correspond to what was observed since the last MS-ADAPT execution. The subscripts i and $i-1$ in the metrics refer, respectively, to the current and previous values.

The optimization phase is carried out with two main cases: (i) evaluating MOB and (ii) evaluating QAR and QQS. These are illustrated on Fig. 8. For case (i) MOB (lines 5 to 9): (i.a) MOB near BOF. There are buckets with outdated interval close to the acceptable limit, thus it is necessary to improve insertion rate. For that sake, the number of threads assigned to insertion (IPP) should be increased, as is performed in line 6. (i.b) MOB is zero. Buckets are up to date and resources assigned to insertion could be assigned to query execution, thus increasing the number of threads assigned to EQP (*increaseEQP* in

Algorithm 5: MS-ADAPT Algorithm

```
1  def ms_adapt(EQP, IPP, BOF, step):
2      Step 1: Monitoring
3      MOB_i, QQS_i, QAR_i ← monitor()
4      Step 2: Optimizing
5      if MOB ≥ BOF then
6          [EQP][IPP] ← decreaseEQP(EQP, IPP, step)
7      else
8          if MOB == 0 then
9              [EQP][IPP] ← increaseEQP(EQP, IPP, IPP)
10     if (QQS_i > QQS_{i-1}) ∨ (QAR_i > QAR_{i-1}) then
11         [EQP][IPP] ← increaseEQP(EQP, IPP, step)
12     if QAR_i < QAR_{i-1} then
13         [EQP][IPP] ← decreaseEQP(EQP, IPP, step)
14     Step 3: Return new conf. for deployment
15     return [EQP][IPP]
```

line 9). The MS-ADAPT considers the number of available threads to be allocated between the levels of parallelism EQP and IPP such that $|EQP| + |IPP| = AvailableThreads$. Thus, every time the algorithm changes one of the values (EQP or IPP), the other value must be updated to maintain their sum equal to the number of available threads. This is observed in the algorithm as the return of increase or decrease functions is a tuple [*EQP*, *IPP*]. Further, there is also an step that configures the size of the change in EQP and IPP every time they are adjusted. The default value of the algorithm is 1, but it could be adjusted accordingly. For instance, in our experiments, we used a step of 5 to reduce the number of possible [*EQP*, *IPP*] configurations and, consequently, experiments necessary in the comparison of MS-ADAPT to static configurations.

As stated, case (ii) observes QAR metric variation: (ii.a) QAR increase (line 11). The systems is experiencing a large number of query requests and, consequently, the query parallelism (EQP) should grow to improve the system throughput. (ii.b) QQS increase (line 11). There is an accumulation in the query task queue. Thus, resources should be assigned to query processing in order to optimize throughput/minimize response times. (c) QAR reduction (line 13). A decrease in the query rate indicates it may be possible to release query processing threads to insertion tasks, enhancing IPP. After computing the [*EQP*, *IPP*] configuration for the current system load, the reconfiguration step begins and threads are reassigned to different processing tasks as they finish their current job.

We wanted to highlight that MS-ADAPT is a heuristic algorithm and, as such, it is was not expected to return the best solution for the current system status in each call. Instead, it was designed as a lightweight solution that is frequently executed to explore online load variations in order to adapt to changes and improve response times. We used this approach because trying to derive the configuration that would lead to the best performance is complex and costly, and would probably offset the gains with the adaptation due to the constant load variations and need to compute that configuration frequently.

7 Experimental evaluation

This section evaluates the performance of our parallel IVFADC. Experiments were carried out using a cluster machine in which each node is equipped with a AMD EPYC 7742 processor with 64 cores at 3.40 GHz and 256GB of RAM. For each compute node, 60 out of 64 cores were bound to computing in a Query Processor (QP) for either query or insertion processing. Thus, MS-ADAPT is allowed to configure the usage of 60 threads for [*EQP*, *IPP*]. The remaining 4 cores of each compute node were statically allocated for communication and other management operations.

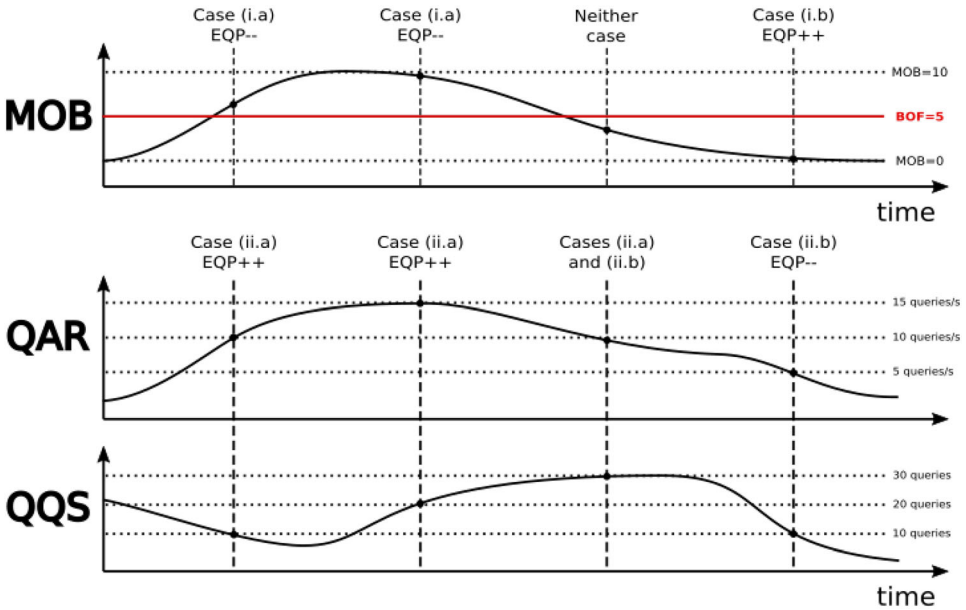


Fig. 8 Illustration of how MS-ADAPT decisions for two different examples are presented. On the first, only the impact of MOB and BOF are displayed (top part). On the second example, both QAR and QQS metrics are evaluated (two bottom graphs). On a given time that MS-ADAPT is executed (vertical dashed lines), it can allocate more query resources (EQP++), allocate more insertion resources (EQP–), or make no changes. Please notice that every time EQP changes, IPP is also modified to maintain all resources in use ($|EQP| + |IPP| = AvailableThreads$)

The experiments used a dataset with up to 344 billion SIFT descriptors. The quality of the search was evaluated based on the recall for the 100 nearest neighbors returned (1-recall@R = 100). Here we are interested in the online setting with varying workloads, which we simulated using *Poisson* distributions. These use a fixed λ for each experiment, but changed between runs to show the system performance under different loads. Thus, we have a Poisson distribution for query arrival rate and another for the insertion arrival rate. In both cases, λ is set with the following 5 query level factors (QLF) and 5 insertion level factors (ILF): 0.2, 0.4, 0.6, 0.8, and 1.0, multiplied by the maximum throughput for query and insertion rates. The maximum throughput was calculated by executing the system in a setting in which all queries or insertions requests were available in the beginning of the experiment (batch). This rate was computed for each number of QPs used.

The experiments considered only cases in which the sum of λ for query and insertion were $\leq$ 1.2. Cases with higher arrival rates are extreme overloads in which queue waiting times would dominate the response times. The experiments executed in Sects. 7.2, 7.3, and 7.4 have used different ILF factors and they correspond to cases with dynamic datasets. In these cases, MS-ADAPT was executed every 0.1 s. We noticed that a executing it in time intervals between 0.1 and 0.5 s led to negligible differences to response times.

In our experimental settings, 11 different static configurations were used as a baseline, varying EQP and IPP in steps of 5 *threads* (e.g., [5][55], [10][50], ..., [55][5]). MS-ADAPT is compared against the static configuration with the best average performance among all QLF $\times$ ILF levels evaluated. Our code is available in.[1]

7.1 IVFADC vs FLANN

This section performs a comparison of IVFADC with FLANN [23] and the exhaustive search. FLANN is based on the *Priority Search K-Means Tree* strategy, and, in turn, the exhaustive search was executed with the Faiss library that uses the BLAS/LAPACK libraries and serves as a solid reference for understanding trade-offs between exact and approximate searches. 1 M SIFT descriptors dataset and 10 K queries were used for this evaluation. Quality was measured using the 1-recall@1 metric [15], which tells us the percentage of query results in which the nearest neighbor is ranked first. The experiments were run sequentially using a single CPU core, and performance is evaluated as the accuracy is varied. In this case, FLANN automatically selects the parameters for each given precision, while w and C_c were modified in IVFADC (shown as w /# of centroids C_c in Fig. 9), along with R (size of the list of k nearest neighbors) needed to achieve a given quality.

The trade-offs between accuracy and performance for IVFADC and FLANN are shown in Fig. 9. As observed, for the same level of accuracy, IVFADC is significantly faster than FLANN. Also, IVFADC uses 26 MB of RAM, while FLANN required about 600 MB of memory. Low memory cost is another important attribute of IVFADC to allow indexing of very large datasets. Also, the exhaustive search took 57 s to run, which is much slower than IVFADC, even when IVFADC is set to achieve high accuracy (e.g. about 95%).

7.2 The impact of MS-ADAPT to response time

This evaluation measures the impact of MS-ADAPT to response time. We employed a Temporal Index with $T = 4$ and $s = 360$ so that the amount of data indexed during the tests do exceed the machine RAM, where T represents the number of temporal partitions and s the time interval in which these partitions are updated. The greater the value of T, the greater the amount of data available for search, and in turn, the greater the s, the faster the process of discarding data. See Fig. 7 for details. IVFADC was configured with $C_c = 4096$ and $w = 16$ that leads to a 1-recall@R = 100 equivalent of 63.7%. The initial index contains 1 billion SIFT descriptors and a single QP node was used. We have chosen these values of C_c and w because they are in the range of values that have shown to attain a reasonable compromise between quality and search time in the original IVFADC paper [15]. Additionally, we have not seen significant differences among strategies for different algorithms' parameters.

The system throughput for cases in which it is submitted only for query processing and insertions was first measured. These rates are, respectively, 126.18 and 9350.15 tasks per second. These values were them used in the rest of experiments to multiply the intensity levels λ to calculate the average of the Poissons used.

The response times using a best static configuration and MS-ADAPT to set [*EQP*, *IPP*] for BOF = 0, which means that the index must be updated for a query to be executed, are presented in Fig. 10. Only the upper part of the matrix with configurations is filled because of the restriction $QLF + ILF \leq 1.2$. Meaning that the system load is limited to 120% of the maximum throughput it may achieve. The configuration [20][40] achieved the best average response time for static cases. It can be observed, as expected, that as the intensity of the tasks streams (QLF and IFL) increase there is a growth in response times. It is specially true for cases in which QLF + IFL is $\geq$ to 1, where the system is overloaded and queuing time dominates the response times.

[1] https://github.com/guineri/msadapt_pqanns.

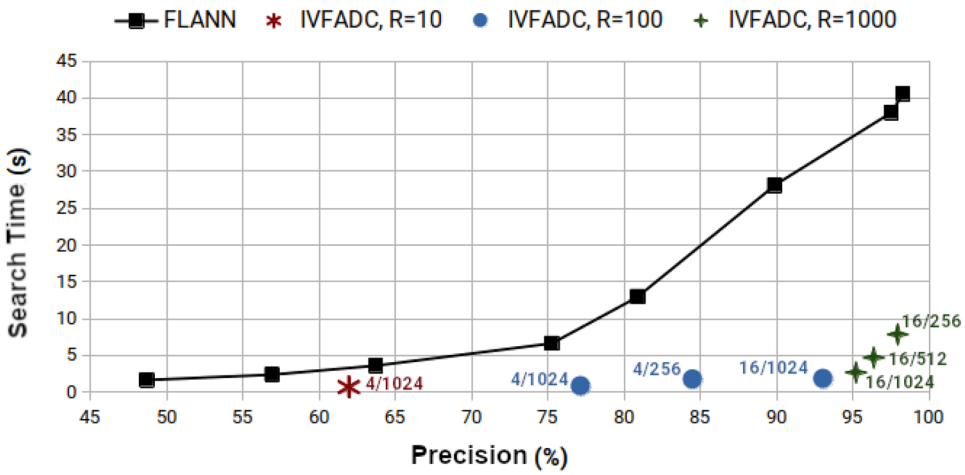


Fig. 9 Comparison between IVFADC and competitors at different search quality levels

Further, MS-ADAPT kept, in all cases, the average response time close to or smaller than the best static configuration. The ability of MS-ADAPT to adapt during the execution enables it to respond to variation of the load factor effectively. It is important to highlight that, in each predefined QLF×ILF combination there are variations in the intensity of the streams given the arrival of the tasks follows a Poisson distribution. That said, even the best static settings can still be a sub-optimal solution. MS-ADAPT is able to adapt to these variations, and in these cases we can see that the average response time was smaller than the best static settings.

7.3 The effect of BOF to response times

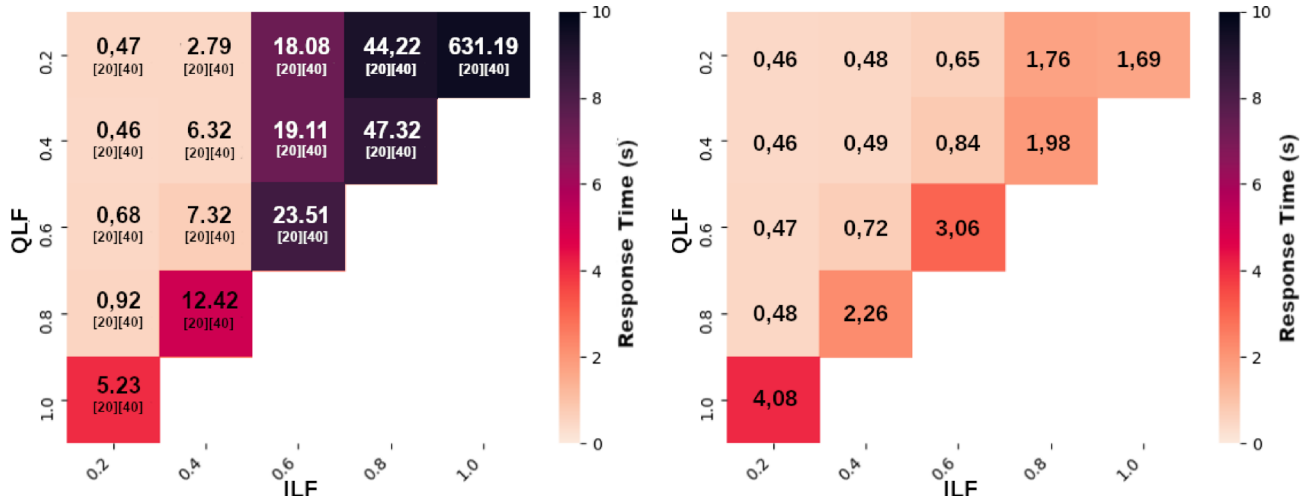
This section evaluates the impact of changes in BOF to the response times. The algorithm configurations are the same used in previous section. Here we have used BOF values of 5 and 10 s, meaning that an index bucket with pending insertions from up to those values can still be used to process queries. The results are presented in Figs. 11 and 12.

The increase in BOF resulted in better response times for all cases due to flexibility added by the scenario. However, it may be also observed that MS-ADAPT has been able to make better use of BOF. For instance, for QLF and IFL configurations summing 1 and 1.2 and BOF = 5 as compared to BOF=0, the static configuration had an average response time reduction of 1.48× and 3.08×, respectively. For the same configurations (QLF+ILF summing 1 and 1.2), MS-ADAPT reduces the average execution time in 9.1× and 32.3× vs the best static setting. The BOF = 10 contributed to slightly smaller response times for high load cases, but the gains as compare to BOF = 5 as smaller than in the migration from BOF = 0 to BOF = 5. This shows that a flexibility to deal with bucket updates is sufficient for MS-ADAPT to accommodate multiple tasks (insertion and query) effectively.

7.4 Scalability of different partitioning strategies

This section evaluates the performance and scalability of our distributed memory IVFADC with different partitioning strategies. We have executed a weak-scaling evaluation in which the size of indexed dataset increases proportionally to the number of compute nodes. This is more appropriate than strong-scaling in our application domain because we have to deal with very large and increasing datasets. The system was executed in the following settings: (i) 40 nodes (8 Co; 32 QPs) and 86 billion SIFT descriptors; (ii) 80 nodes (16 Co; 64 QPs) and 172 billion SIFT descriptors; (iii) 160 nodes (32 Co; 128 QPs) and 344 billion SIFT descriptors. The IVFADC was configured to use $C_c = 8192$ and $w = 8$, which resulted in a 1-recall@100 of 58.3%. The values of C_c and w were modified as compared to previous experiments because we used a large per QP node dataset here, thus we wanted to have similar execution times and recall as before to enable large-scale runs without very high resource utilization. Multiple partitioning strategies were employed: DES, BES, SABES, and SABES++. In addition, three different data stream intensity scenarios were considered: ILF = 0.2, 0.4 and 0.6 (all using QLF = 1.0). For comparative purposes, the parallelization strategy described in [29] was executed under the same conditions using the IVFADC as the baseline search algorithm.

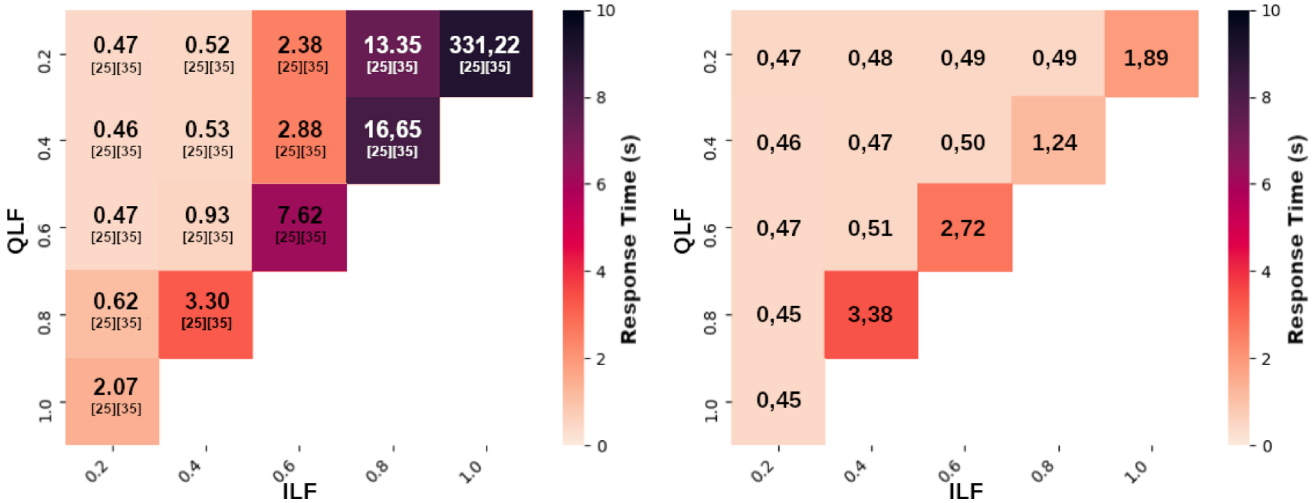
Figure 13 shows the experimental results. As observed, the scalability of the distributed memory machines is superlinear for BES, SABES, and SABES++. Interestingly, as the intensity of the insertion stream grows, the impact on the reduction of the query throughput is more intense in the SABES distribution strategy. This aspect is



(a) Best Static configuration: 22 and 40 threads, respectively, for query an insertion processing.

(b) MS-ADAPT

Fig. 10 Comparison between best static configuration and MS-ADAPT using BOF = 0



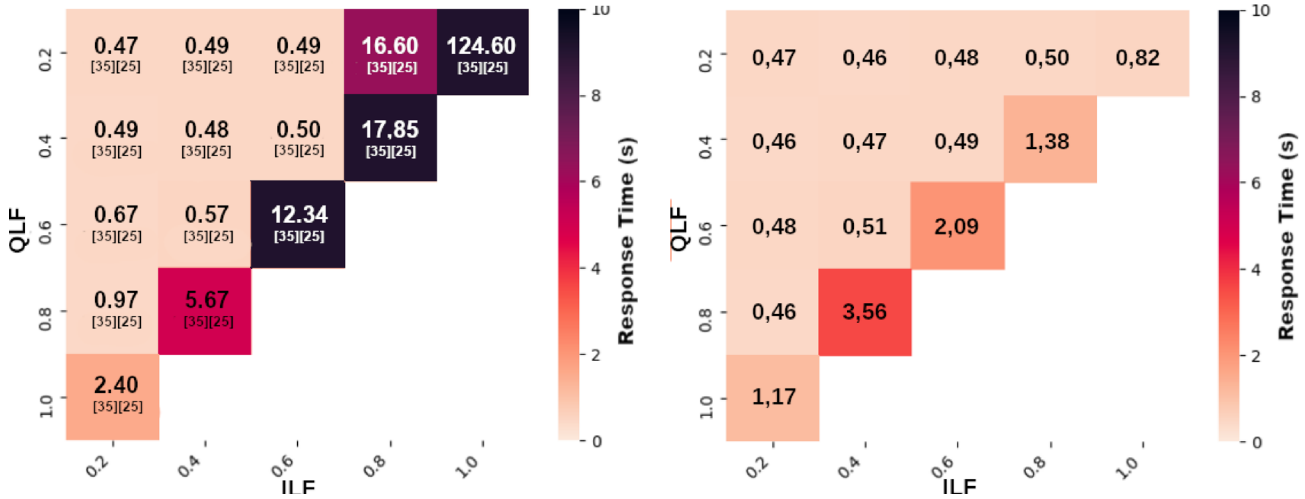
(a) Best Static configuration: 25 and 35 threads, respectively, for query an insertion processing.

(b) MS-ADAPT

Fig. 11 Comparison between baseline and MS-ADAPT using BOF = 5

directly related to the data imbalance in this approach, which increases when more descriptors are indexed. Furthermore, the [29] parallelization has similar behavior of DES partitioning, as it essentially benefits from equal data partitioning as well. However, the difference in performance between from DES is due to the fact that the insertions, in the [29] solution, happen in a data structure (Delta Table) separate from the structure in which queries are computed. This aspect eliminates synchronizations necessary in cases when query and insertion takes place in the same data structure. The [29] solution proposes aggregations (between Delta and Static Tables) with a greater volume of data in more spaced time intervals, which shows us that, when the data distribution is equal, this mechanism tends to be more efficient as insertions and queries happen on the same index.

The superlinear scalability attained by BES, SABES, and SABES++ is a consequence of accessing a smaller number of QPs to answer a query on these cases. For instance, when 32 QPs are employed, DES uses all 32 QPs, as expected, while BES and SABES need an average of only 12.7 and 4.3 QPs, respectively, to respond to a query. This aspect consolidates the assertion that by distributing *buckets* close in space in the same Query Processor, as proposed by SABES and SABES++, we are increasing the search locality, which will happen in only a subset of QPs.



(a) Best Static configuration: 35 and 25 threads, respectively, for query an insertion processing.

(b) MS-ADAPT

Fig. 12 Comparison between baseline and MS-ADAPT using BOF = 10

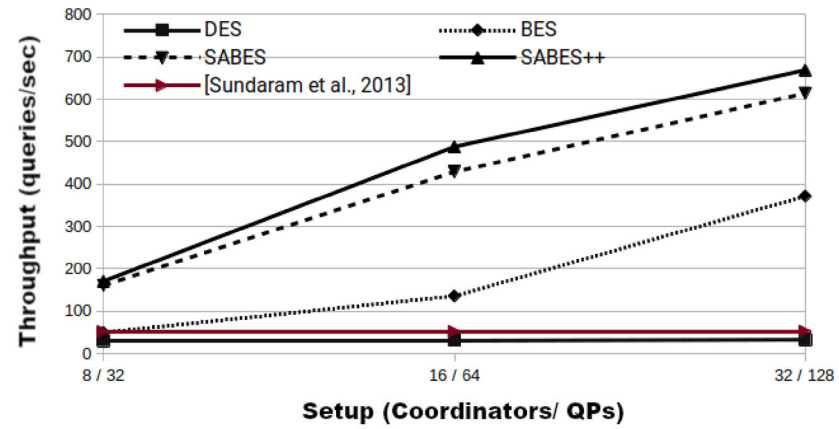
The SABES++ that reduces the imbalance among QPs has been able to significantly improve SABES in all cases. For instance, for the case with $QLF = 1.2$ and $IFL = 0.4$, SABES++ throughput is about 1.2× higher than that attained by SABES. Data balancing in SABES and SABES++ is shown in Table 2 which presents the maximum and minimum number of descriptors indexed by QPs in both approaches. There is a significant reduction in the imbalance of data with SABES++.

8 Conclusions and future directions

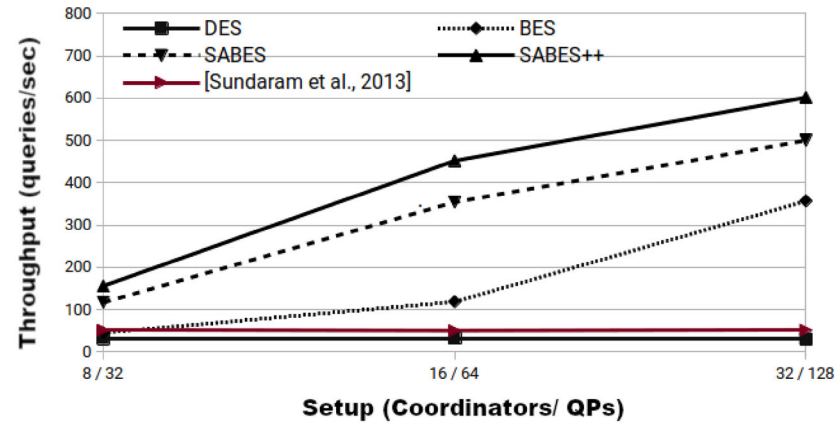
CBMR applications are becoming popular in several online services. These services have to deal with very large and increasing datasets, which require the use of distributed memory computing systems to match their processing demands. New challenges are faced in this scenario due to the dynamic dataset and load characteristics as queries should be processed on-the-fly as new data are inserted.

In this work, we address these challenges with parallelization strategies and a run-time system with optimization to deal with online CBMR services. Our approach enables concurrent processing of queries and data insertions in a stream fashion, and also optimizes the use of computing resources targeting to reduce query response time in scenarios where query/insertion rates vary during the execution. This is achieved with an algorithm for adapting resource allocated to deal with the query/insertion streams, called MS-ADAPT. MS-ADAPT identifies variations in intensity of the streams and chooses the best configuration of resource allocation (CPU cores) to be used. The results show that static solutions for this problem are suboptimal given the variability of load of queries and insertions streams, while MS-ADAPT can adapt at run-time reaching average response time close to or better than the best static configuration in several scenarios evaluated.

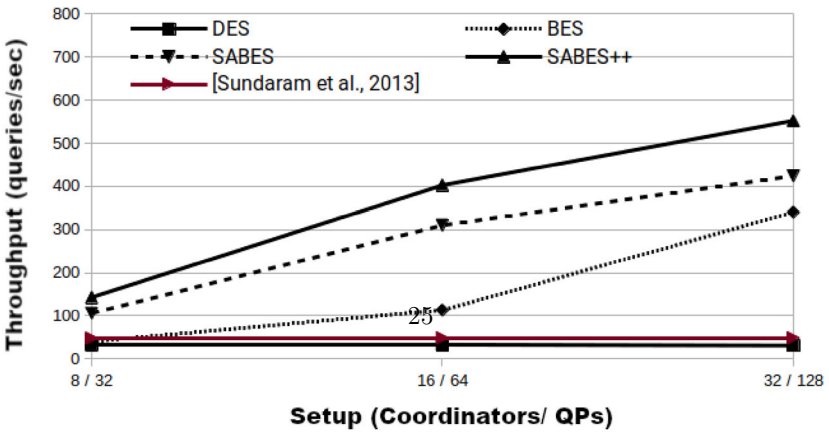
For the future work we intend to enable the use of GPUs in the search to further improve the throughput attained by our solutions. This study will also allow us to understand the impact of this hardware to each parallelization strategy as we believe the gains may be different with each approach. In fact, we expect that spatial-aware strategies may attain higher speedups as the data required to answer a search are located in a few nodes. The current version of the SABES++ strategy performs the balanced centroids distribution with a clustering computed before execution. However, in dynamic datasets, data imbalance may appear during the execution as new data items are indexed. As such, it may be interesting to further evaluate this aspect and study strategies that could reconfigure the data distribution at run-time. Additionally, while MS-ADAPT attained reasonable gains compared to the static strategy, we intend to investigate other strategies. The main goal will be aimed at the trade-off between better allocations and time to compute such solutions. Finally, we also want to evaluate data partition solutions to deal with heterogeneous machines. The heterogeneity could be in different components of the machine as the processor, memory capacity, attached storage technology etc. These settings are more likely to be found in cloud environments.



(a) QLF=1.0 e ILF=0.2



(b) QLF=1.0 e ILF=0.4



(c) QLF=1.0 e ILF=0.6

Fig. 13 Query throughput of IVFADC in a *weak scaling* evaluation, where data partitioning strategies and intensity of the insertion stream were varied

Table 2 Data distribution using SABES and SABES++

Strategy	# of QPs	Data distribution		
		Min	Max	Std
SABES	4	167,078,445	336,612,032	89,578,841
	32	14,249,858	62,618,917	12,678,393
SABES++	4	160,689,301	390,357,008	75,624,558
	32	21,451,654	46,859,624	4,549,251

The minimum and maximum quantity of objects (descriptors) assigned to a QP and the standard deviation across all 32 QPs

Author contributions All authors contributed to the study conception and design. Material preparation, data collection and analysis were performed by GA and WBJ. The first draft of the manuscript was written by GA and GT and all authors commented on previous versions of the manuscript. All authors read and approved the final manuscript.

Funding This work was supported in part by National Council of Scientific and Technological Development (CNPq), Fundação de Amparo à Pesquisa do Estado de Minas Gerais (FAPEMIG), Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), PRPq/UFMG.

Data Availability Enquiries about data availability should be directed to the authors.

Declarations

Conflict of interest The authors have no relevant financial or non-financial interests to disclose.

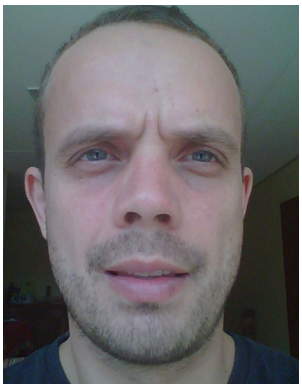
References

1. Böhm, C., Berchtold, S., Keim, D.A.: Searching in high-dimensional spaces: index structures for improving the performance of multimedia databases. ACM Comput. Surv. **33**, 322–373 (2001). https://doi.org/10.1145/502807.502809
2. Amato, F., Greco, L., Persia, F., et al.: Content-Based Multimedia Retrieval, pp. 291–310. Springer International Publishing, Cham (2015)
3. Sitaula, C., Shahi, T.B., Marzbanrad, F., et al.: Recent advances in scene image representation and classification. Multimed. Tools Appl. (2023). https://doi.org/10.1007/s11042-023-15005-9
4. Dujaili, M.J.A.: Survey on facial expressions recognition: databases, features and classification schemes. Multimed. Tools Appl. (2023). https://doi.org/10.1007/s11042-023-15139-w
5. Khunsongkiet, P., Bootkrajang, J., Techawut, C.: Low-level feature image retrieval using representative images from minimum spanning tree clustering. Multimed. Tools Appl. (2023). https://doi.org/10.1007/s11042-023-15605-5
6. Wan, J., Wang, D., Hoi, S.C.H., et al.: Deep learning for content-based image retrieval: a comprehensive study. In: Proceedings of the 22Nd ACM International Conference on Multimedia. ACM, New York, NY, USA, MM '14, pp. 157–166 (2014). https://doi.org/10.1145/2647868.2654948
7. Douze, M., Jégou, H., Sandhawalia, H., et al.: Evaluation of GIST descriptors for web-scale image search. In: Proceedings of the ACM International Conference on Image and Video Retrieval. ACM, New York, NY, USA, CIVR '09, pp. 19:1–19:8 (2009). https://doi.org/10.1145/1646396.1646421
8. Jégou, H., Perronnin, F., Douze, M., et al.: Aggregating local image descriptors into compact codes. IEEE Trans. Pattern Anal. Mach. Intell. **34**(9), 1704–1716 (2012). https://doi.org/10.1109/TPAMI.2011.235
9. Zezula, P., Amato, G., Dohnal, V., et al.: Similarity Search: The Metric Space Approach, vol. 32. Springer Science & Business Media, Berlin (2006)
10. Friedman, J.H., Bentley, J.L., Finkel, R.A.: An algorithm for finding best matches in logarithmic expected time. ACM Trans. Math. Softw. **3**(3), 209–226 (1977). https://doi.org/10.1145/355744.355745
11. Beygelzimer, A., Kakade, S., Langford, J.: Cover trees for nearest neighbor. In: Proceedings of the 23rd International Conference on Machine learning, ICML '06, pp. 97–104 (2006). https://doi.org/10.1145/1143844.1143857
12. Muja, M., Lowe, D.G.: Scalable nearest neighbor algorithms for high dimensional data. IEEE Trans. Pattern Anal. Mach. Intell. **36**(11), 2227–2240 (2014). https://doi.org/10.1109/TPAMI.2014.2321376
13. Weber, R., Schek, H.J., Blott, S.: A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In: Proceedings of the 24rd International Conference on Very Large Data Bases. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, VLDB '98, pp. 194–205 (1998). https://doi.org/10.5555/645924.671192
14. Gionis, A., Indyk, P., Motwani, R.: Similarity search in high dimensions via hashing. In: Proceedings of the 25th International Conference on Very Large Data Bases. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, VLDB '99, pp. 518–529 (1999). https://doi.org/10.5555/645925.671516
15. Jegou, H., Douze, M., Schmid, C.: Product quantization for nearest neighbor search. IEEE Trans. Pattern Anal. Mach. Intell. **33**(1), 117–128 (2010). https://doi.org/10.1109/TPAMI.2010.57
16. Andrade, G., Teodoro, G., Ferreira, R.: Scalable and efficient spatial-aware parallelization strategies for multimedia retrieval. In: 32nd IEEE International Symposium on Computer Architecture and High Performance Computing, SBAC-PAD 2020, Porto, Portugal, September 9–11, 2020, pp. 124–131. IEEE (2020). https://doi.org/10.1109/SBAC-PAD49847.2020.00027
17. Andrade, G., Ferreira, R., Teodoro, G.: Spatial-aware data partition for distributed memory parallelization of ANN search in multimedia retrieval. Parallel Comput. **115**, 102992 (2023). https://doi.org/10.1016/j.parco.2022.102992
18. Stupar, A., Michel, S., Schenkel, R.: RankReduce—processing K-nearest neighbor queries on top of MapReduce. In: Proceedings of the 8th Workshop on Large-Scale Distributed Systems for Information Retrieval (LSDS-IR'10), pp. 1–6 (2010). http://ceur-ws.org/Vol-630/lsdsir2.pdf
19. Bahmani, B., Goel, A., Shinde, R.: Efficient distributed locality sensitive hashing. In: Proceedings of the 21st ACM International Conference on Information and Knowledge Management (CIKM), pp. 2174–2178 (2012). https://doi.org/10.1145/2396761.2398596
20. Moise, D., Shestakov, D., Gudmundsson, G., et al.: Indexing and searching 100M images with Map-reduce. In: Proceedings of the 3rd ACM Conference on International Conference on Multimedia Retrieval, ICMR '13, pp. 17–24 (2013). https://doi.org/10.1145/2461466.2461470
21. Souza, R., Fernandes, A., Teixeira, T.S.F.X., et al.: Online multimedia retrieval on CPU-GPU platforms with adaptive work partition. J. Parallel Distrib. Comput. **148**, 31–45 (2021). https://doi.org/10.1016/j.jpdc.2020.10.001

22. Malkov, Y.A., Yashunin, D.A.: Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. IEEE Trans. Pattern Anal. Mach. Intell. **42**(4), 824–836 (2020). https://doi.org/10.1109/TPAMI.2018.2889473
23. Muja, M., Lowe, D.: Fast approximate nearest neighbors with automatic algorithm configuration. In: VISAPP 2009—Proceedings of the 4th International Conference on Computer Vision Theory and Applications, vol. 1, pp. 331–340 (2009)
24. Santini, S.: A meta-indexing method for fast probably approximately correct nearest neighbor searches. Multimed. Tools Appl. **81**(21), 30465–30491 (2022). https://doi.org/10.1007/s11042-022-12690-w
25. Chávez, E., Marroquín, J.L., Navarro, G.: Fixed queries array: a fast and economical data structure for proximity searching. Multimed. Tools Appl. **14**(2), 113–135 (2001). https://doi.org/10.1023/A:1011343115154
26. Kruliš, M., Skopal, T., Lokoč, J., et al.: Combining CPU and GPU architectures for fast similarity search. Distrib. Parallel Databases **30**(3), 179–207 (2012). https://doi.org/10.1007/s10619-012-7092-4
27. Teodoro, G., Valle, E., Mariano, N., et al.: Approximate similarity search for online multimedia services on distributed CPU-GPU platforms. VLDB J. **23**(3), 427–448 (2014). https://doi.org/10.1007/s00778-013-0329-7
28. Andrade, G., Fernandes, A., Gomes, J.M., et al.: Large-scale parallel similarity search with product quantization for online multimedia services. J. Parallel Distrib. Comput. **125**, 81–92 (2019). https://doi.org/10.1016/j.jpdc.2018.11.009
29. Sundaram, N., Turmukhametova, A., Satish, N., et al.: Streaming similarity search over one billion tweets using parallel locality-sensitive hashing. Proc. VLDB Endow. **6**(14), 1930–1941 (2013). https://doi.org/10.14778/2556549.2556574
30. Meisburger, N., Shrivastava, A.: Distributed tera-scale similarity search with MPI: provably efficient similarity search over billions without a single distance computation. CoRR abs/2008.03260 (2020). arXiv:2008.03260
31. Johnson, J., Douze, M., Jégou, H.: Billion-scale similarity search with GPUs. IEEE Trans. Big Data **7**(3), 535–547 (2021). https://doi.org/10.1109/TBDATA.2019.2921572
32. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008). https://doi.org/10.1145/1327452.1327492
33. Forum, M.P.: MPI: a message-passing interface standard. Technical report, USA (1994)
34. Gudmundsson, G.T., Jónsson, B.T., Amsaleg, L., et al.: Prototyping a web-scale multimedia retrieval service using spark. ACM Trans. Multimed. Comput. Commun. Appl. (2018). https://doi.org/10.1145/3209662
35. Zaharia, M., Chowdhury, M., Franklin, M.J., et al.: Spark: cluster computing with working sets. In: Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing, HotCloud'10, p. 10 (2010)
36. Doshi, I., Das, D., Bhutani, A., et al.: LANNS: a web-scale approximate nearest neighbor lookup system. Proc. VLDB Endow. **15**(4), 850–858 (2021). https://doi.org/10.14778/3503585.3503594
37. Zhu, N., Lu, Y., He, W., et al.: Towards update-efficient and parallel-friendly content-based indexing scheme in cloud computing. Int. J. Semant. Comput. **12**(2), 191–213 (2018). https://doi.org/10.1142/S1793351X1840010X
38. Singh, A., Subramanya, S.J., Krishnaswamy, R., et al.: Freshdiskann: a fast and accurate graph-based ANN index for streaming similarity search. CoRR abs/2105.09613 (2021). arXiv:2105.09613
39. Gil-Costa, V., Marin, M.: Load balancing query processing in metric-space similarity search. In: 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID 2012), pp. 368–375 (2012). https://doi.org/10.1109/CCGrid.2012.30
40. Yang, K., Wang, H., Du, M., et al.: An efficient indexing technique for billion-scale nearest neighbor search. Multimed. Tools Appl. (2023). https://doi.org/10.1007/s11042-023-14825-z
41. Kerdprasop, K., Kerdprasop, N., Sattayatham, P.: Weighted K-means for density-biased clustering. In: Tjoa, A.M., Trujillo, J. (eds.) Data Warehousing and Knowledge Discovery. Springer, Berlin, Heidelberg (2005)
42. Wei, X., Liu, Y., Wang, X., et al.: A survey on quality-assurance approximate stream processing and applications. Future Gener. Comput. Syst. **101**, 1062–1080 (2019). https://doi.org/10.1016/j.future.2019.07.047
43. Lev, Y., Luchangco, V., Olszewski, M.: Scalable reader-writer locks. In: Proceedings of the Twenty-First Annual Symposium on Parallelism in Algorithms and Architectures, pp. 101–110 (2009)

Guilherme Andrade holds a degree in Computer Science from the Federal University of São João del Rei (2012) and a Master's Degree in Computer Science at Federal University of Minas Gerais (2014). He currently is a PhD candidate in Computer Science from the Computer Science Department (DCC) at the Federal University of Minas Gerais, working on research in the areas of high performance computing in heterogeneous architectures.

Willian Barreiros Jr. holds a degree in Computer Engineering from the University of Brasília (2016) and a Master's Degree in Computer Science at University of Brasília (2018). He currently is a PhD candidate in Computer Science from the Computer Science Department at the Federal University of Brasília, working on research in the areas of high performance computing in heterogeneous architectures.



Leonardo Rocha is an Associated Professor at Computer Science Department at Federal University of São João Del Rei, Brazil. He holds a PhD in Computer Science from Federal University of Minas Gerais, Brazil (2009). His research interests include Information Retrieval, Data Mining, Machine Learning and Recommender Systems, having published about 200 papers in these areas.

Renato Ferreira is a full professor in the Department of Computer Science at Universidade Federal de Minas Gerais. His research focuses on compiler and run-time support for high performance computing and large, dynamic datasets. It involves both high performance, important issue from the applications end-users' perspective, and high-level programming abstractions, important for the application domain developers.

George Teodoro received his M.S. and Ph.D. degrees in Computer Science from the Universidade Federal de Minas Gerais (UFMG), Brazil, in 2006 and 2010. He is currently an assistant professor in the Computer Science Department at the University of Brasilia (UnB), Brazil. His primary areas of expertise include high performance runtime systems for efficient execution of biomedical and data-mining applications on distributed heterogeneous environments.