AnalyticDB-V: A Hybrid Analytical Engine Towards Query Fusion for Structured and Unstructured Data

Chuangxian Wei, Bin Wu, Sheng Wang, Renjie Lou,
Chaoqun Zhan, Feifei Li, Yuanzhe Cai
Alibaba Group
{chuangxian.wcx,binwu.wb,sh.wang,json.lrj,lizhe.zcq,lifeifei,yuanzhe.cyz}
@alibaba-inc.com

ABSTRACT

With the explosive growth of unstructured data (such as images, videos, and audios), unstructured data analytics is widespread in a rich vein of real-world applications. Many database systems start to incorporate unstructured data analysis to meet such demands. However, queries over unstructured and structured data are often treated as disjoint tasks in most systems, where *hybrid queries* (*i.e.*, involving both data types) are not yet fully supported.

In this paper, we present a hybrid analytic engine developed at Alibaba, named *AnalyticDB-V* (ADBV), to fulfill such emerging demands. ADBV offers an interface that enables users to express *hybrid queries* using SQL semantics by converting unstructured data to high dimensional vectors. ADBV adopts the *lambda* framework and leverages the merits of approximate nearest neighbor search (ANNS) techniques to support hybrid data analytics. Moreover, a novel ANNS algorithm is proposed to improve the accuracy on large-scale vectors representing massive unstructured data. All ANNS algorithms are implemented as physical operators in ADBV, meanwhile, accuracy-aware cost-based optimization techniques are proposed to identify effective execution plans. Experimental results on both public and in-house datasets show the superior performance achieved by ADBV and its effectiveness. ADBV has been successfully deployed on Alibaba Cloud to provide hybrid query processing services for various real-world applications.

PVLDB Reference Format:
Chuangxian Wei, Bin Wu, Sheng Wang, Renjie Lou, Chaoqun Zhan, Feifei Li, Yuanzhe Cai. AnalyticDB-V: A Hybrid Analytical Engine Towards Query Fusion for Structured and Unstructured Data. *PVLDB*, 13(12): 3152-3165, 2020.
DOI: https://doi.org/10.14778/3415478.3415541

1. INTRODUCTION

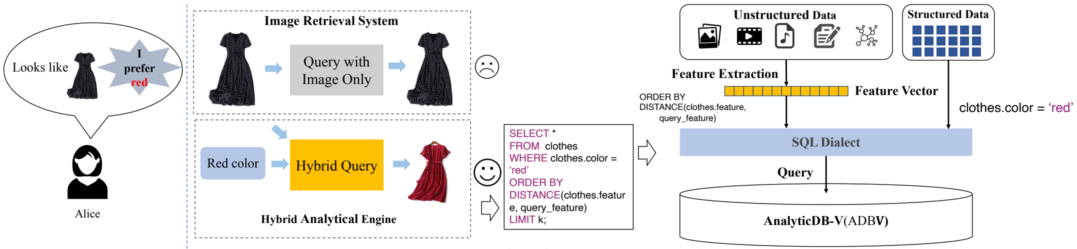
Massive amounts of unstructured data, such as images, videos, and audios, are generated each day due to the prevalence of smartphones, surveillance devices, and social media apps. For example, during the 2019 Singles' Day Global Shopping Festival, up to 500PB unstructured data are ingested into the core storage system at Alibaba. To facilitate analytics on unstructured data, content-based retrieval systems [45] are usually leveraged. In these systems, each piece of unstructured data (*e.g.*, an image) is first converted into a high dimensional feature vector, and subsequent retrievals are conducted on these vectors. Such vector retrievals are widespread in various domains, such as face recognition [47, 18], person/vehicle re-identification [56, 32], recommendation [49], and voiceprint recognition [42]. At Alibaba, we also adopt this approach in our production systems.

Although content-based retrieval system supports unstructured data analytics, there are many scenarios where both unstructured and structured data shall be jointly queried (we call them *hybrid queries*) for various reasons. First, a query over unstructured data may be inadequate to describe the desired objects, where a *hybrid query* helps improve its expressiveness. For instance, on e-commerce platform like Taobao, one potential customer may search for a dress with conditions on price (less than $100), shipment (free-shipping), rating (over 4.5), and style (visually similar to a dress worn by a movie star). Second, the accuracy of state-of-the-art feature vector extraction algorithms is far from satisfactory, especially on large datasets, where a *hybrid query* helps to improve the accuracy. For example, the false-negative rate of face recognition increases by 40 times when the number of images scales from 0.64 million to 12 million [14]. Therefore, imposing constraints on structured attributes (such as gender, age, image capturing locale, timestamp in this context) can narrow down the vector search space and effectively improve the accuracy. In summary, the *hybrid query* is of great value to a vast number of emerging applications.

However, most existing systems do not provide native support for *hybrid queries*. Developers have to rely on two separate engines to conduct hybrid query processing: a vector similarity search engine ([25, 8, 54]) for unstructured data and a database system for structured data. This practice has inherent limitations. First, we have to implement extra logic and post-processing step atop two systems to ensure data consistency and query correctness. Second, *hybrid queries* cannot be jointly optimized as sub-queries are executed on two engines independently.

To address this challenge, we design and implement a new analytical engine, called *AnalyticDB-V* (ADBV) inside the OLAP system AnalyticDB (ADB) [53] at Alibaba Cloud, that manages massive feature vectors and structured data

Proceedings of the VLDB Endowment, Vol. 13, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3415478.3415541



```
SELECT *
FROM  clothes
WHERE clothes.color =
'red'
ORDER BY
DISTANCE(clothes.featur
e, query_feature)
LIMIT k;
```

Figure 1: Hybrid query example.

and natively supports *hybrid query*. During the design and development of this system, we have encountered and addressed several vital challenges:

Real-time management of high-dimensional vectors. The extracted feature vectors from unstructured data are usually of extremely high dimensions. For example, at Alibaba, vectors for unstructured data can reach 500+ dimensions in many scenarios, such as online shopping applications. In addition, these vectors are being generated in real-time. The real-time management (*i.e.*, CRUD operations) on such high-dimensional vectors is burdensome for existing databases and vector search engines. On one hand, online database systems with similarity search support (*e.g.*, PostgreSQL and MySQL) only works for vectors of up to tens of dimensions. On the other hand, vector similarity search engines (such as Faiss) adopt `ANNS` (Approximate Nearest Neighbor Search) approaches [33, 23] to process and index high-dimensional vectors in an offline manner, which fail to handle real-time updates.

Hybrid query optimization. *Hybrid query* lends new opportunities for joint execution and optimization considering both feature vectors and structured attributes. However, the hybrid query optimization is inherently more complex than existing optimizations. Classical optimizers that support top-k operations [29, 20, 19] do not have to consider the accuracy issue, *i.e.*, all query plans lead to identical execution results. However, for *hybrid queries*, approximate results are returned by `ANNS` (on the vectors) to avoid exhaustive search, and hence the accuracy of top-k operations varies with the choice of `ANNS` methods and parameter settings. There remains a non-trivial task to balance the quality of approximated results and the query processing speed.

High scalability and concurrency. In many of our production environments, vectors are managed at extremely large scales. For example, in a smart city transportation scenario, we have to manage more than 11.7 billion roads or vehicle snapshots with 100 million newly inserted records each day. Moreover, at least 5 thousand queries need to be processed per second, of which more than 90 percent are *hybrid queries*. In another application scenario of `Freshippo` supermarket, a digitized retail store of Alibaba Group, 800 million 512-dimensional vectors are stored in ADBV. The peak load is 4000 queries per second (QPS), and above 80% of the queries are *hybrid queries*. A distributed architecture is essential for such large-scale workloads. Besides, fast retrieval on massive vectors and fast indexing of new ingested data must be sustained.

In ADBV, we address the above challenges and make major contributions as follows:

- *A real-time analytic engine for hybrid queries.* We present an analytical engine that natively supports *hybrid query* for fusion of structured and unstructured data with real-time updates. To fulfill the real-time requirement, we adopt the *lambda* framework with different `ANNS` indexes for the streaming layer and the batching layer. The neighborhood-based `ANNS` methods in the streaming layer support real-time insertions but consume a lot of memory. The encoding-based `ANNS` methods in the batching layer consume much less memory, but require offline training before construction. *Lambda* framework can periodically merge newly ingested data from the streaming layer into the batching layer.
- *A new ANNS algorithm.* For the sake of improving the accuracy on large-scale vectors representing massive unstructured data, a novel `ANNS` index, called **V**oronoi **G**raph **P**roduct **Q**uantization (`VGPQ`), is proposed. This algorithm could efficiently narrow down the search scope in the vector space compared to `IVFPQ` [23] with margi-nal overhead. According to the empirical study, `VGPQ` is more effective for fast indexing and queries on massive vectors than `IVFPQ`.
- *Accuracy-aware cost-based hybrid query optimization.* In ADBV, `ANNS` algorithms are wrapped as physical operators. Hence, we can rely on query optimizer to efficiently and effectively support *hybrid query* processing. Physical operators in a relational database always return exact results. However, these newly introduced physical operators may not strictly follow relational algebra, and output approximate results instead. Due to the nature of approximation, we deliver new optimization rules to achieve the best query efficiency. These rules are naturally embedded in the optimizer of ADBV.

In the following sections, we will provide details of ADBV. §2 introduces the background of *hybrid query* and SQL dialects. §3, §4, and §5 present our designs and implementations, *i.e.*, overall system design, vector processing (`ANNS`) algorithms, and accuracy-aware cost-based hybrid query optimization respectively. Experimental evaluations are conducted in §6. Related works are discussed in §7, and the conclusion is drawn in §8.

2. BACKGROUND

2.1 Motivation

To accurately retrieve records of interest, a typical *hybrid query* consists of both similarity constraints on feature vectors (extracted from unstructured data) and value constraints on structured data. Consider the example shown in Figure 1, the target is to retrieve dresses that are visually similar to the query image, but in red color. Conventionally, these two kinds of constraints are handled by two separate systems. The developer needs to query for top-k images using a vector search engine (such as Faiss [25, 8], vearch [30]), and at the same time retrieve the color information from a database. After that, records obtained from both systems are conjunctively merged to derive final results.

Such practice induces extra development efforts and computational cost. It may happen that less than k records (returned from a vector search engine) successfully pass the color or style constraint expressed in the user query, and thus a top-k result cannot be constructed to meet the quantity requirement explicitly mentioned in user query. Hence, the developer has to carefully set the number of records to be retrieved by the vector search engine. Also, the execution efficiency has vast potential to be optimized. For example, if only a small percent of clothes fulfill the structured constraints (*i.e.*, red color), it will be more efficient to retrieve records with the structured constraints first, and then identify closest feature vectors from the retrieved set directly. ADBV is therefore motivated to solve above problems.

ADBV allows users to express a *hybrid query* as a SQL statement, and executes it efficiently without manual tuning. Note that both unstructured and structured data can be stored within one table. In particular, unstructured data is transformed to vectors (via feature extraction functions) at insertion stage and stored in a column.

2.2 SQL dialects

ADBV provides a flexible and easy-to-use SQL interface. Developers can easily port their applications to ADBV with minimal effort.

2.2.1 SQL statements

Create Table. Tables are created in a manner that is similar to standard SQL, except for feature vector related operations. The feature columns are defined as follows: `DistanceMeasure` defines the distance function used for `ANNS`, in which Squared Euclidean, Dot Product, and Hamming distance are all supported. `ExtractFrom` defines the way that a feature vector is extracted (see §2.2.3). Users can specify the `ANNS` index on a feature vector, as shown below:

```
ANN INDEX feature_index (column_feature)
```

Insert. The following statements illustrate the syntax of the insert. If the `ExtractFrom` keyword is used previously during table creation, the feature vector will be generated automatically according to the definition. Alternatively, the value of a feature vector can be inserted explicitly in the format of an array.

```
--insert implicitly
INSERT INTO table_name(C$_1$, C$_2$,···,C$_n$)
VALUES(v$_1$, v$_2$, ···,v$_n$);
--insert explicitly
INSERT INTO table_name(C$_1$, C$_2$,···,C$_n$,
    column_feature)
VALUES(v$_1$, v$_2$, ···,v$_n$,
array[$e_1$,$e_2$,$e_3$,...,$e_k$]::float[]);
```

Select. The following SQL syntax illustrates the usage of queries on unstructured data. `unstructured_data` is the original object (*e.g.*, sentences of a text, URL of an image). Usually, unstructured data is stored in an online storage service (*e.g.*, Object Storage Service, Azure Blob Storage) and referred to as a URL. ADBV can read the object from the URL and extract the corresponding feature vector using `FEATURE_EXTRACT` function. `DISTANCE` represents the vector distance function defined at the table creation stage.

```
SELECT *
FROM table_name
WHERE table_name.C$_1$ = '$v_1$'
ORDER BY DISTANCE(table_name.column_feature,
FEATURE_EXTRACT('unstructured_data'))
LIMIT k;
```

Delete. The delete statement is the same as standard SQL, except that feature vector similarity constraints can also be applied as a filter.

2.2.2 A running example

Here we go through the entire process of issuing a *hybrid query* using the example in Figure 1. We first create a table that stores vector features and other attributes for clothes via the following SQL:

```
CREATE TABLE clothes(
id int,
color varchar,
sleeve_t varchar,
···,
image_url varchar,
feature float[512]
COLPROPERTIES (
DistanceMeasure = 'SquaredEuclidean',
ExtractFrom = 'CLOTH_FEATURE_EXTRACT(image_url)'))
PARTITION BY ClusterBasedPartition(feature);
```

ClusterBasedPartition is a partition function that maps the feature vectors of the same cluster into the same partition (detailed in §3.3). Then we insert one record into the table, which represents a red dress:

```
--insert implicitly
INSERT INTO clothes(id,color,sleeve_t,···,image_url)
VALUES(10001,'red','long',···,"protocal://xxx.jpg");
--insert explicitly
INSERT INTO clothes(id,color,sleeve_t,···,image_url)
VALUES(10001,'red','long',···,"protocal://xxx.jpg",
array[1.1,3.2,2.4,...,5.2]::float[]);
```

Finally, we retrieve the target dress (*i.e.*, in red color and similar to the query image) using the following SQL:

```
SELECT *
FROM clothes
WHERE clothes.color = 'red'
ORDER BY DISTANCE(clothes.feature,
CLOTH_FEATURE_EXTRACT('protocal://xxx.jpg'))
LIMIT k;
```

2.2.3 Vector extraction UDF

ADBV currently supports feature extraction models for a variety of data sources, including faces, clothes, vehicles, documents, *etc.* The feature extraction models adopt the state-of-the-art deep learning models trained on large-scale datasets. Moreover, the user defined function (UDF) API is also opened to users, which allows uploading their own vector extraction models and vector extraction functions to ADBV.

3. SYSTEM DESIGN

ADBV is built on top of a PB-scale OLAP database system named *AnalyticDB* [53] from Alibaba, which relies on two fundamental components, *i.e.*, Pangu [2] for reliable and permanent distributed storage, and Fuxi [55] for resource management and computation job scheduling. ADBV enhances AnalyticDB to incorporate vectors and support hybrid queries. In this section, we present key system designs that improve the functionality and effectiveness of vector management and hybrid query processing.

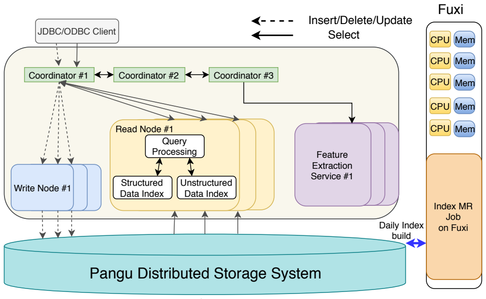


Figure 2: ADBV architecture

3.1 Architecture overview

The architecture of ADBV is presented in Figure 2, which is mainly composed of three types of nodes: *Coordinator*, *Write Node*, and *Read Node*. *Coordinators* accept, parse, optimize SQL statements, and dispatch them to read/write nodes. ADBV adopts a typical read/write decoupling approach [53], which trades consistency for low query latency and high write throughput. Hence, *write nodes* are only responsible for write requests (*i.e.*, `INSERT`, `DELETE` and `UPDATE`), while *read nodes* are for `SELECT` queries. Newly ingested data is flushed into Pangu [2] upon committed. ADBV adopts the *lambda* framework in the storage layer (§3.2) to manage vectors efficiently: the streaming layer deals with real-time data insertion and modification; and the batching layer periodically compresses newly inserted vectors and rebuilds `ANNS` indexes. Moreover, ADBV pushes down several expensive predicates to the storage layer so as to fully utilize the computation capability of those nodes.

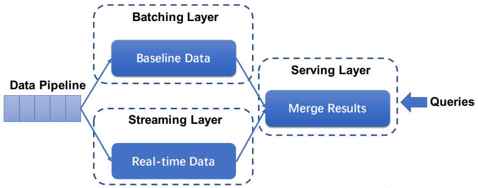


Figure 3: Lambda framework in ADBV.

3.2 Lambda framework

Since the complexity of searching over the entire vector dataset is unacceptable, an index must be built to mitigate the cost. However, traditional index techniques like KD-tree [6] and ball-tree [38] widely used in low dimensions do not work well for high-dimensional vectors generated by deep learning models. It has been empirically proved that such solutions exhibit linear complexity for high-dimensional vectors [51]. Therefore, algorithms such as `HNSW` (Hierarchical Navigable Small World) [34] and `LSH` (Locality-Sensitive Hash) [12] are proposed for real-time index building[1] on vectors in an approximate manner. However, existing algorithms fail to either handle large data volumes owing to large memory consumption, or provide results with sufficient accuracy. Taking `HNSW` as an example, it requires both the index data and feature vectors stored persistently in memory to avoid disk I/O, otherwise its performance will significantly degenerate. Beside the raw data, each record requires around 400 bytes in memory for its index.

We adopt the *lambda* framework to solve the challenge of supporting real-time inserts. Under this framework, ADBV uses `HNSW` to build index for newly-inserted vectors (*i.e.*, incremental data) in real time. Periodically, ADBV merges incremental data with baseline data into a global index (§3.2.2) according to the proposed `VGPQ` algorithm (§4.2) and discards the `HNSW` index.

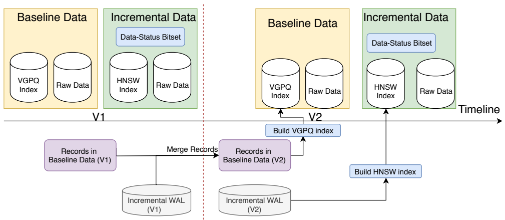


Figure 4: Baseline Data and Incremental Data in ADBV

3.2.1 Overview

Figure 3 depicts the *lambda* framework, which consists of three layers: batching layer, streaming layer, and serving layer. These layers work jointly to process each incoming query. The batching layer returns search results based on baseline data (discussed in §3.2.2). For the streaming layer, it performs two tasks: processing data modification (*i.e.*, `INSERT`, `DELETE`, and `UPDATE`), and producing search results over incremental data. For `SELECT` statements, partial results from batching and streaming layers are merged by coordinators to derive final results. The serving layer is responsible for issuing requests to both batching and streaming layers and returning results to clients. Different layers reside in different types of nodes, *i.e.*, serving layer in coordinators, batching layer in read nodes, and streaming layer both read and write nodes.

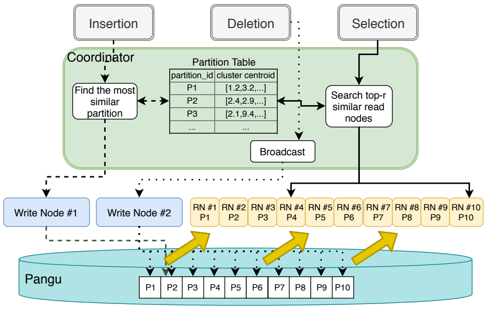
3.2.2 Baseline and Incremental Data

To better support the *lambda* framework, ADBV contains two types of data: baseline data and incremental data, as shown in Figure 4. The incremental data contains newly-written WALs (kept in Pangu), as well as vector data and its indexes on read nodes. It also contains a data-status bitset to track which vector data has been deleted. Incremental raw data and indexes are much smaller in size compared to baseline data, and can be entirely cached in memory. We use `HNSW` to build the index for incremental data, so that index building and search can be conducted at the same time. The baseline data contains the entire set of historical raw data and indexes, which are stored in Pangu. As baseline data could be massive, we apply `VGPQ` (§4.2) to build index asynchronously to sustain memory efficiency.

When new data continuously arrive, searching on incremental data slows down due to the huge memory consumption of `HNSW`. Hence, an asynchronous merging process is launched to merge incremental data into baseline data periodically, as shown in Figure 4. During this process, current incremental data is marked as immutable (*i.e.*, old-version), and a new version of incremental data is created to handle subsequent write requests. Then, the old-version incremental data and the baseline data are merged into a new version of baseline data, in which the index is rebuilt using `VGPQ` to replace the `HNSW` index.

Before the merge process completes, queries are served by the old-version baseline data and both versions of incremental data. After it is done, we serve subsequent queries with new-versions of baseline and incremental data, and safely discard their old versions. Note that vectors marked as

[1] Real-time index building here means the system can continually build index for newly-inserted vector data and be queried subsequently.



partition_id	cluster centroid
P1	[1.2,3.2,...]
P2	[2.4,2.9,...]
P3	[2.1,9.4,...]
...	...

Figure 5: Clustering-based partition pruning

deleted in incremental data are removed from baseline data during the merge.

3.2.3 *Data Manipulation*

Here we present how INSERT, UPDATE, DELETE, SELECT are processed in the three layers of the *lambda* framework over baseline and incremental data. For INSERT, the streaming layer appends incoming data to the incremental data and constructs the corresponding index. When DELETE arrives, the streaming layer does not really remove the vector data, but marks it as deleted in the data-status bitset. The UPDATE operation is performed as a combination of DELETE and INSERT: mark the original vector data as deleted in the data-status bitset and append the updated record into the incremental data. More details are discussed in [53].

For SELECT statements, they are first sent to both streaming layer and batching layer, which search over their manged data respectively (both using indexes). After that, the results returned from two layers are merged and further filtered by the data-status bitset to get rid of deleted records. Finally, the serving layer returns the records that have the highest similarity to the query.

3.3 Clustering-based partitioning

As presented in Figure 5, ADBV provides the ability to partition vector data across a number of nodes to achieve high scalability. However, partition techniques for structured data, such as hash and list partition, are unsuitable for vector data. It is because they rely on equivalence and range assessment, while analytic on vector data is based on similarity (*e.g.*, Euclidean distance). A direct adoption of these strategies will require queries to execute on all partitions indiscriminately without pruning effect.

To solve this problem, we propose a clustering-based partition approach. For the partitioned column, we apply k-means [17] to calculate centroids according to the number of partitions. For example, if we define 256 partitions, 256 centroids will be calculated. Each vector is clustered to the centroid with the largest similarity (*e.g.*, smallest Euclidean distance), and hence each cluster forms a partition. Index building and data manipulation (§3.2) are then conducted on each individual partition. As for partition pruning, ADBV dispatches the query to N partitions, which have the largest similarity to the queried vector. N is a query hint defined by users, which reflects the trade-off between query performance and accuracy.

Note that the partitioning on vector data is disabled by default, since the partitioning on structured data is the first choice in ADBV. Users can enable it when query performance are of high importance. When a partitioned column is defined on vector data, ADBV will sample a portion of records, calculate centroids, and re-cluster (re-distribute) the entire set of records accordingly.

4. VECTOR PROCESSING ALGORITHMS

Hybrid query processing relies heavily on dedicated approximate nearest neighbor search algorithms. In ADBV, efficient ANNS algorithms are implemented as physical operators to handle top-k retrieval tasks. In this section, we introduce how ANNS algorithms are used to process vector queries, and propose a novel ANNS algorithm named VGPQ, *i.e.*, Voronoi Graph Product Quantization, to further improve query efficiency (in the batching layer).

4.1 Vector query processing

The objective of vector query processing is to obtain top-k nearest neighbor *w.r.t.* the input vector query q of dimension d. This can be viewed as a well-known *nearest neighbor search* problem, where hundreds of algorithms have been proposed in the literature. The baseline option is a brute-force search on the entire dataset to get top-k vectors. It returns exact results, and the time complexity acceptable when n is relatively small. Many of tree-based algorithms are proposed to divide the original search space into subspaces recursively and build the corresponding index for subspaces to reduce the time complexity greatly [16, 43]. However, these algorithms are no better than exhaustive search on high dimensional features based on observations of [51]. In order to perform fast retrieval on high dimensional datasets, we need to trade off query accuracy for lower running time. Many approaches have been proposed to find approximate nearest neighbors , particularly for high dimensional data in §7. Among them, ADBV primarily leverages neighborhood-based and quantization-based algorithms to do nearest neighbor search on data in the streaming layer and batching layer respectively.

At the streaming layer, ADBV implements HNSW (as mentioned in §3.2). HNSW is a neighborhood-based ANNS algorithm that utilizes the neighborhood information of each vector point towards other points and relies on the small-world navigation assumption [27]. It supports dynamic insertion, but cannot scale to large datasets. Therefore, HNSW is suitable to support querying on newly-inserted data.

At the batching layer, ADBV applies a quantization-based algorithm PQ [23] to encode vectors. Its main idea is to represent raw vector data with a compact encoding, *i.e.*, low dimensional and lossy, to reduce the expensive pairwise distance calculation cost by a constant factor. However, the encoding codebook of PQ should be trained offline. Hence, we deploy this method to support querying on large-scale accumulated baseline data. In order to avoid scanning all vectors' PQ codes for each incoming query point, IVFPQ [23] uses k-means to cluster PQ codes into groups at index construction stage, and only scans the most relevant groups at query stage. Based on IVFPQ, we propose VGPQ to further improve the efficiency (§4.2) and use it instead.

4.2 Voronoi graph product quantization

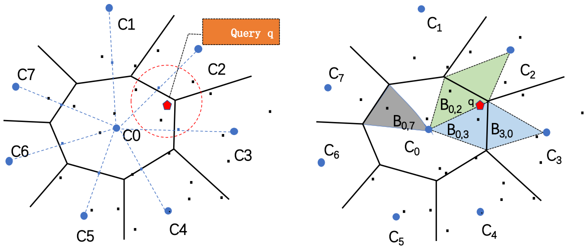
Main idea of VGPQ. Recall that IVFPQ clusters vectors into groups via k-means and each then vector's PQ code is characterized by the corresponding centroid. An incoming vector query is redirected to the most relevant groups, where relevant codes are scanned. In VGPQ, we build a voronoi diagram from these centroids in IVFPQ on the original space, and partition each voronoi cell into several subcells based on following heuristics. For the ease of explanation, we denote C as a centroid, and refer to the centroid and the corresponding voronoi cell interchangeably. Given the query vector q in Figure 6(a), its top-3 neighbors (highlighted by the red circle) should be returned as a result. Since no prior knowledge is provided, IVFPQ traverses C_0 (which contains two target points) and all seven adjacent cells in order to find the third target point in C_2. To solve this problem, we draw a line between centroid C_0 (also called *anchor* centroid) and all adjacent centroids, and then mark the midpoint of each line segment implicitly, as shown in Figure 6(a). Now we can divide the original cell C_0 into seven subcells based the distance to seven midpoints, respectively. Likewise, subcells are generated in the same manner for all centroids. Here we denote each subcell as $B_{i,j}$, where i, j indicates the anchor centroid C_i and neighbor centroid C_j. We denote the distance between q and subcell $B_{i,j}$ as $d(q, B_{i,j})$, which is defined as the distance between q and the midpoint of line segment between C_i and C_j. For the query vector q in Figure 6(b), among these subcells, q is closer to the midpoints of line segments (C_0, C_2) and (C_0, C_3). That is to say, we only need visit 6 subcells (*i.e.*, $B_{0,2}$,$B_{2,0}$,$B_{0,3}$,$B_{3,0}$,$B_{2,3}$,$B_{3,2}$) to obtain a sufficient number of ground-truth neighbors for q.

Preprocessing. As outlined in Algorithm 1, the VGPQ index is built through three steps. Firstly, k-means clustering is applied to partition the vector space. Secondly, neighbors are selected for each centroid and each cell is divided into subcells accordingly. Finally, PQ codes for vectors are inserted into associated inverted files (§4.3). In order to reduce the cost of voronoi graph construction, we choose top-b nearest centroids to build corresponding subcells instead.

Algorithm 1: BUILD_VGPQ($\mathcal{D}$, k, b)

```
Input: vector dataset $\mathcal{D} = \{v_0, v_1, \ldots, v_n\}$, number
       of centroids $n\_clusters$, number of subcells
       $n\_subcells$
Output: VGPQ index
$\mathcal{C} \leftarrow$ find the $n\_clusters$ centroids from $\mathcal{D}$ by
  k-means, where $\mathcal{C} = \{c_0, c_1, \cdot, c_k\}$;
for $i \leftarrow 0, 1, \ldots, |\mathcal{C}|$ - 1 do
    $nb(c_i) \leftarrow$ find $n\_subcells$ closest centroids in $\mathcal{C}$;
for $i \leftarrow 0, 1, \ldots, |\mathcal{D}|$ - 1 do
    $c_p \leftarrow$ find the nearest centroid $i$-th vector point
      from $\mathcal{C}$;
    $B_{p,q} \leftarrow$find the nearest subcell in $nb(c_p)$;
    compute the product quantization code of $v_i$;
    append product quantization code into $B_{p,q}$
      associated storage structure;
```

Query processing. After the VGPQ index is built, ANNS queries can be answered following above main idea. First, s anchor centroids are located, from each of which b closest subcells are selected as candidates respectively. VGPQ further filters $s \times b$ candidates according to their distances to q. After that, distances between q and all vectors in candidate subcells are computed. VGPQ returns top-k vectors with highest similarities to users, where k is the value defined by the LIMIT keyword in the query statement.



(a) IVFPQ (b) VGPQ

Figure 6: Motivation of VGPQ

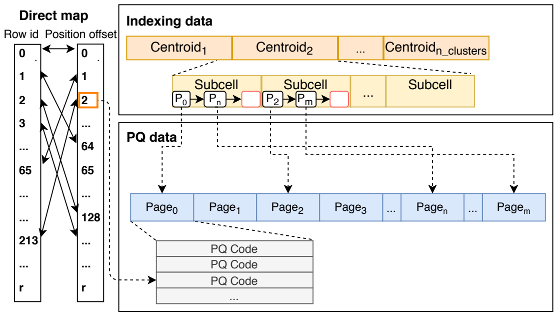


Figure 7: Storage of VGPQ index

Compared with IVFPQ, VGPQ divides a voronoi cell into subcells by incorporating neighborhood information between centroids, which reduces the search space by a constant factor. Empirical evaluation results are provided in §6 to validate our design.

4.3 Storage design for VGPQ

From a system perspective, we design the in-memory storage structure to facilitate VGPQ execution. It mainly consists of three components: Indexing data, PQ data, and Direct map (see Figure 7). PQ data is responsible for storing PQ codes. It occupies a consecutive memory space divided into pages. Each page is assigned with a unique *pageid* and contains a fixed number of PQ codes belonging to a particular subcell. In practice, the size of a page is often set as a multiple of the I/O block size to improve the data ingestion efficiency as well as the disk data loading performance.

Indexing data stores an array of feature vectors, and each tuple of it corresponds to one anchor centroid. In VGPQ, we divide every anchor centroid's space into b subregions. PQ codes belonging to a subregion are organized into pages in PQ data. Accordingly, each anchor centroid will be associated with b subcells in Indexing data, and each subcell contains a linked list of *pageid*s pointing to pages in PQ data. Once a page is filled during index construction stage, its *pageid* will be attached to the tail of the corresponding linked list in Indexing data. In ADBV, each vector point is given a unique *rowid* as its identity, which helps to fuse query on structured and unstructured data.

In addition, we develop a bi-directional map between the *rowid* and the position of its PQ code in PQ data. Given a *rowid*, Direct map indicates where the PQ code is stored in PQ data, and vice versa. On the one hand, we only need store one copy of PQ code for each record in ADBV. Any ANNS method works on PQ code could access it with the aid of Direct map. On the other hand, it helps VGPQ to handle *hybrid queries*, and we will discuss details in §5.1.

5. HYBRID QUERY OPTIMIZATION

In this section, we discuss the design details of hybrid query execution and optimization in ADBV. Firstly, ANNS algorithms are conceptually treated as database indexes and accessed by *scan* operators in physical plans. These newly designed operators will be smoothly injected into existing query execution plans. After that, the optimizer of ADBV will enumerate multiple valid physical plans for each input *hybrid query*, and the optimal one will be determined by the usage of a cost model proposed in §5.2. At last, we discuss how to determine underlying hyperparameters closely related to query accuracy. As mentioned in §3.2 ADBV only retains a small quantity of (newly inserted) vector data in the streaming layer, whose query processing time is negligible compared to it in the batching layer. Hence, we focus on query execution and optimization for the batching layer.

To better elaborate our designs, two typical selection query examples are used throughout this section: a simple vector search query $Q1$ and a *hybrid query* $Q2$, where $T = (id, c, f)$ is a table containing structured columns id, c and a unstructured column f. For *hybrid queries* like $Q2$, complex structured predicates are naturally supported in ADBV. For the sake of brevity, we only list two structured predicates in *where* clause to show the main idea. Structured column related predicates are evaluated and processed in an exact manner. Thus, corresponding physical operators will not influence approximation quality of final results. In others words, above mentioned hyper parameter tuning logic is particularly designed for ANNS related operators. Additionally, we assume the B-tree index is already built on c for illustration purpose.

```
SELECT id, DISTANCE(f, FEATURE_EXTRACT('img'))
        as distance
FROM T
ORDER BY distance,
-- return top-k closest tuples
LIMIT k;
```

Q1: vector query example

```
SELECT id, DISTANCE(f, FEATURE_EXTRACT('img'))
        as distance
FROM T
--structured predicates
WHERE T.c >= p1 and T.c <= p2
ORDER BY distance,
-- return top-k closest tuples
LIMIT k;
```

Q2: hybrid query example

5.1 Hybrid query execution

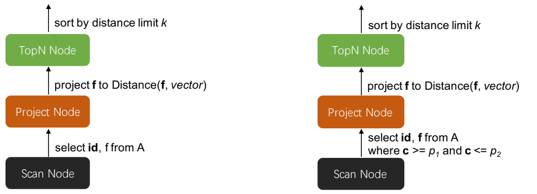
ADBV follows the classical way to translate these two queries into corresponding logical execution plans firstly, as shown in Figure 8. Since each value in a high-dimensional unstructured column f often takes over 2000 bytes at storage, reducing the number of vector read operations from disk could significantly improve the query processing efficiency. Hence, we push down similarity search with regards to the input *img* to storage nodes, in which only 'neighbor' vectors need be popped.

The optimizer of ADBV analyzes the Abstract Syntax Tree delivered by the query parser to detect a pattern representing the operation to return top-k unstructured column related tuples (*i.e.*, "order by DISTANCE() LIMIT k"). If the corresponding pattern is found, the optimizer will translate the logical plan to multiple physical execution plans that perform nearest neighbor searches on unstructured columns. Apart from the conventional brute-force search, ANNS algorithm are wrapped into *anns scan* nodes, which can be directly injected into existing physical plans to obtain an approximate set of nearest neighbors *w.r.t.* the feature point extracted from the query image. In this way, ANNS algorithms help to reduce the expensive computation cost from retrieving top-k neighbors in a large dataset. Given characteristics of different nearest neighbor search algorithms, four effective physical plans are proposed and discussed over the query example $Q2$, as illustrated in Figure 9.

Plan A (*brute-force search*). *Plan A* is a basic and conservative solution for *hybrid queries*. As shown in Figure 9, right after the *B-Tree scan*, all tuples that satisfy structured predicates (*i.e.*, $c \geq p_1$ and $c \leq p_2$) are popped from the storage layer. Then, brute-force search will be conducted to obtain exact top-k neighbors at the read node. This plan only works well on small datasets (*e.g.*, thousands of points). Otherwise, it leads to unacceptable processing time due to its linear complexity. It is also our conservative strategy by default when other plans involving *anns scan* operators fail to satisfy user's accuracy requirements.

Plan B (*PQ Knn Bitmap Scan*). If approximate results satisfy the accuracy requirement and the number of tuples returned by a structured index scan reaches a certain level (*e.g.*, ten thousands of points), ADBV tries to insert an *anns scan* node into physical plans to speedup the nearest neighbor search. Among these plans, *Plan B* is the cleanest one. After performing a *B-Tree Scan* with regards to column c, a set of *rowids* referring to the qualified tuples are collected. Then we build a bitmap over this tuple set. As ADBV adopts a column store, remaining columns of a tuple can be easily fetched given its *rowid*. *PQ Knn Bitmap Scan* obtains the pre-computed PQ codes corresponding to *rowids* one by one, and performs asymmetric distance computation (ADC) [23] to estimate squared distances against the input image feature. After that, a set of candidate nearest neighbors are gathered and reported to the upper layer in ascending order of calculated distances. Instead of reporting exact k *rowids*, *PQ Knn Bitmap Scan* amplifies it to $\sigma * k$ ($\sigma > 1$). The σ is a hyperparameter to improve the approximation quality of final results, and we will discuss its tuning process in §5.3. *Plan B* leverages the succinct encoding and the fast distance computation to reduce the cost of nearest neighbor search. In practice, the compression ratio of PQ encoding usually reaches 16:1.

Plan C (*VGPQ Knn Bitmap Scan*). When the number of records satisfying structured predicates reach a higher level (*e.g.*, larger than one million), *Plan B* is incapable of returning candidates within seconds. Instead, *anns scan* operator based on VGPQ, *i.e.*, *VGPQ Knn Bitmap Scan*, is considered by the optimizer. It adds bitmap tests to the original VGPQ as to remove the redundant computation between input image feature and those PQ codes that fail to meet structured predicates. Recall that in the storage of VGPQ, a bi-directional map between *rowid* and corresponding addresses in PQ data is maintained in Direct map. Given the address of one PQ code, we can find its *rowid* in the Direct map. When its *rowid* is not in the bitmap, we directly skip to the next one. The entire execution process of *Plan C* is very similar to *Plan B*: we first get a bitmap of *rowids* using *B-Tree Scan*; and then perform *VGPQ Knn Bitmap Scan* to get an approximate nearest neighbors set with size $\sigma * k$. Moreover, in the execution of *Plan C*, we need not only tune the σ,


sort by distance limit *k*
TopN Node
project **f** to Distance(**f**, *vector*)
Project Node
select **id**, f from A
Scan Node
sort by distance limit *k*
TopN Node
project **f** to Distance(**f**, *vector*)
Project Node
select **id**, **f** from A
where **c** >= p_1 and **c** <= p_2
Scan Node

(a) $Q1$ Logical Plan (b) $Q2$ Logical Plan

Figure 8: Logical plans of query examples

but also the VGPQ related hyperparameters (*e.g.*, the visited subcell ratio).

Plan D (*VGPG Knn Scan*). In order to obtain exact results, conventional optimizer that supports the top-k operation [29, 20, 19] does not introduce any optimization rule to change the execution order between *Filter Operator* with *Top-N Operator* in the query execution. In other words, *Filter Operator* should be performed before *Top-N Operator*. Nevertheless, the approximation nature of *hybrid queries* relieves the rigorous constraint on the order of these two operators. *Plan D* is formulated after this exchange rule is applied. When most records in a large table meet structured predicates, the cost of filter operation in all above plans will become a bottleneck. For example, fetching a large number of qualified tuples from the B-Tree index becomes unexpectedly prohibitive. Hence, *Plan D* allows the optimizer to exchange the execution order of *Filter Operator* and *Top-N Operator*. We apply the original VGPQ, implemented as *VGPQ Knn Scan*, at the first step, and then verify the approximate result set against the structured predicates (*i.e.*, $c \geq p_1$ and $c \leq p_2$). Since only $\sigma * k$ tuples are left, the computation cost of predicate checking becomes trivial. The execution speed of *Plan D* is also closely related to the hyperparameters of VGPQ.

5.2 Cost model for optimization

In order to identify the optimal plan (among four plans above) for different scenes, we propose a accuracy-aware cost model for *hybrid query* optimization. All Notations used by the cost model are listed in Table 1.

Table 1: Notations used by hybrid query optimization

Notation	Meaning
n	the total number of tuples in database
α	the ratio between n and the number of records satisfying structured predicate
β	the visited subcells ratio during VGPQ index searching process in VGPQ Knn Bitmap Scan
γ	the visited subcells ratio during VGPQ index searching process in VGPQ Knn Scan
$\sigma_{\{B,C,D\}}$	amplification factors of *ANNS Scan* operators in $Plan\{B, C, D\}$
c_1	the total time cost to fetch a vector and compute pairwise distance
c_2	the total time cost to fetch a PQ code and run ADC

Plan A starts from a structured index scan with cost T_0, and $\alpha \times n$ records are supposed to be qualified. Then, similarities between the query vector and qualified vectors are computed via DISTANCE function. The total cost of *plan A* is formulated in Equation 1:

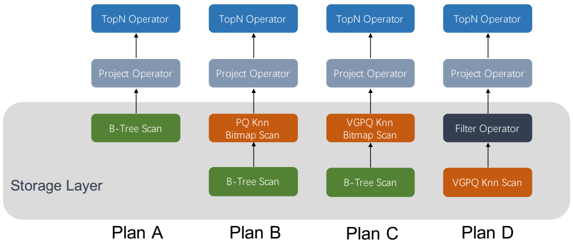


Figure 9: Physical plans of $Q2$

$$cost_A = T_0 + \alpha \times n \times c_1 \quad (1)$$

Plan B first performs the structured index scan, and then the approximate squared distances between the query vector and each vector's PQ code are estimated with ADC respectively. This step finds $\sigma_B \times k$ records to narrow down the brute-force search scope. The total cost of *plan B* is formulated in Equation 2:

$$cost_B = T_0 + \alpha \times n \times c_2 + \sigma_B \times k \times c_1 \quad (2)$$

In *Plan C*, *VGPQ Knn Bitmap Scan* is executed after the structured index scan. The total cost of *Plan C* is formulated in Equation 3.

$$cost_C = T_0 + \beta \times n \times \alpha \times c_2 + \sigma_C \times k \times c_1 \quad (3)$$

In *Plan D*, we switch the execution order of predicate filter and vector index scan in contrast with *Plan C*. The input size is significantly trimmed for predicate filter, whose running time becomes negligible. The total time cost of *Plan D* is formulated in Equation 4:

$$cost_D = \gamma \times n \times c_2 + \sigma_D \times k \times c_1 \quad (4)$$

Given a *hybrid query*, plan execution costs can be calculated with above equations, and then the optimizer chooses the one with minimal cost as the final plan. In addition, the optimizer also needs to tune the choice of β, γ, and σ, which directly affect the query accuracy. This tuning process is discussed in the following section.

5.3 Accuracy-aware hyperparameter tuning

Given the user-defined accuracy requirement and k, we need to find appropriate hyperparameters (*i.e.*, β, γ, and σ) for each possible physical plan under different structured predicates (*i.e.*, α). Otherwise, the optimizer is not able to compute the cost using above presented cost models. However, we can hardly provide a formula with theoretical guarantees to derive the optimal setting. Several heuristic methods are used instead for the hyperparameter tuning. The hyperparameter tuning process consists of two steps, *i.e.*, pre-process step and execution step.

At the pre-process step, we divide the range of α into many disjoint segments, each of which is treated as a bin and recognized by its lowest value. For each plan, we use a grid search method (similar to auto tune in Faiss [25]) to enumerate all combinations of hyperparameters for all bins. Among them, we only consider combinations that could return results satisfying the accuracy requirement. Then the combination of the lowest running time will be recorded as the final setting of the corresponding plan and bin. In ADBV, the pre-process step is carried out after the newly ingested data is merged into the batching layer in *lambda* framework.

At the execution step, for each *ad-hoc* query, we could estimate the value of α' with sophisticated selectivity estimation techniques [40]. These algorithms are already developed in the query optimizer of modern database systems [48, 39, 53]. Since ADBV is built on top of AnalyticDB, we calibrate it in virtue of its query optimizer. Then, we select the pre-tuned combination of hyperparameters belonging to the corresponding bin for each plan. Finally, we identify the most efficient plan for execution by computing corresponding cost models with selected hyperparameter combinations. In practice, these combination candidates work well for most user queries. For a certain plan, if the optimizer fails to pick a candidate that achieves satisfied accuracy from pre-tuned combinations, it will simply ignore this plan in later decision-making phase. In the worst case, when *Plan B,C,D* all fail, the optimizer can still pick *Plan A* for execution.

6. EXPERIMENTS

In this section, we evaluate ADBV on both public and in-house datasets to verify the effectiveness of proposed designs, as well as its performance improvement over existing solutions.

6.1 Experimental setup

Testbed. We conduct the experiments with a 16-node cluster on Alibaba Cloud; each node has 32 logical cores, 150GB DRAM and 1TB SSD. The host machines are all equipped with one Intel Xeon Platinum 8163 CPU (2.50GHz), and hyperthreading is triggered. Machines are connected via a 10Gbps Ethernet network.

Datasets. We use two public datesets and one in-house dataset to evaluate our system, as listed below:

- `Deep1B` [5] is a public dataset consists of image vectors extracted from a deep neural network. It contains **one billion 96**-dimensional vectors.
- `SIFT1B` [24] is a public dataset consists of handcrafted SIFT features. It contains **one billion 128**-dimensional vectors.
- `AliCommodity` consists of **830 million 512**-dimensional vectors extracted from commodity images used at Alibaba. It also contains 21 structured columns including `color`, `sleeve_type`, `style`, `create_time`, *etc.*

These datasets differ in data sources (two public image datasets and one in-house commodity image dataset), feature extraction methods (SIFT and deep neural networks) and vector dimension (96, 128 and 512). Moreover, so far, there is no publicly available dataset that contains both structured data and unstructured data. The in-house hybrid dataset `AliCommodity` is adopted to evaluate hybrid query processing techniques.

In the distributed experiments, these datasets are split into a number of partitions. The subscript of a dataset mentioned below denotes the corresponding partition number (*e.g.*, `SIFT1B_8p` means `SIFT1B` is split into eight partitions). By default, we first shuffle vectors in a dataset, and then evenly distribute them across a number of nodes to build partitions. Otherwise, the partition scheme will be explicitly presented.

Query types. We reuse the two query templates ($Q1$ and $Q2$) provided in §5.1 for evaluation. According to our observations in production environments, most of queries processed by ADBV follow these patterns.

Implementation. *AnalyticDB* [53], an OLAP database system developed in `Java`, is adopted as the backbone of ADBV. `ANNS` algorithms, along with the storage, are primarily written in `C` to obtain optimal performance from advanced optimization techniques (*e.g.*, vectorized processing). These key components of ADBV are accessed through Java Native Interface. In §6.2, we adopt the `C++` implementation of `IVFPQ` provided by Faiss [25].

Metrics. We use the recall to measure the accuracy of a result set returned by an `ANNS` algorithm (or a system). Suppose the exact result set is S, the recall is defined as $recall = \frac{|S \cap S'|}{|S|}$, where S' is the result set returned by a `ANNS` algorithm (or a system) and $|\cdot|$ computes the carnality of a set. We also use the recall in Top-N results (*recall@TopN*) to evaluate the system performance, which means $|S| = |S'| = N$.

At each test, we issue 10,000 SQL requests (from all two types in a round-robin manner) to calculate the average recall and response time. For each query, the query image *img* is sampled from the original dataset uniformly at random. The default value of k is set to 50. We vary the values of p_1 and p_2 on column c to generate test queries with different *selectivity*, which is defined as

$$s = 1 - \frac{number\ of\ tuples\ pass\ the\ predicates}{total\ the\ number\ of\ tuples}. \qquad (5)$$

Let c_{max} and c_{min} be the maximum and minimum value of c, if we assume the values are uniformly distributed between c_{max} and c_{min}, the selectivity could be calculated by $s = 1 - \frac{p_2 - p_1}{c_{max} - c_{min}}$. In order to generate a sample query with the specified selectivity s, we coarsely follow this formula to approach the selectivity at first, then we manually tweak $p1$ or $p2$ to finally get the a query with desired selectivity.

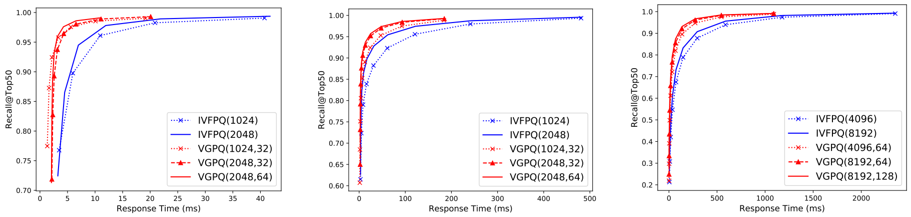
6.2 VGPQ

In this section, we compare `VGPQ` and `IVFPQ` from three aspects, i.e., accuracy, index construction time, and the size of index files. Since the setting of underlying `PQ` encoding affects the size of index files, same settings are always deployed for both algorithms. Note that each method can be built with different parameters, and we indicate used parameters in a bracket after the method name: the first denotes the number of centroids; and the second denotes the number of subcells (is only applicable for `VGPQ`). For example, `VGPQ` (4096, 64) denotes that, in `VGPQ`, 4096 clusters are built and each cluster is subsequently divided into 32 subcells.

Table 2: The construction time and index size comparison between VGPQ and IVFPQ on AliCommodity.

method	time (min)	size (GB)
IVFPQ(4096)	155	112
IVFPQ(8192)	199	112
VGPQ(4096,64)	144	112
VGPQ(8192,64)	178	112
VGPQ(8192,128)	182	112

Table 2 lists the construction time and the size of the index file with different parameter settings on `AliCommodity`. It exhibits that the size of the index file generated by `VGPQ` and `IVFPQ` are very close. Their index construction time mainly depend on the number of centroids in k-means (*n_clusters* in Algorithm 1). With the same the number of centroids, the construction time of `VGPQ` is 10% lower that of `IVFPQ`. The performance of `VGPQ` is also affected by the number of subcells (*n_subcells* in Algorithm 1). However, we observe



(a) SIFT1B (b) Deep1B (c) AliCommodity

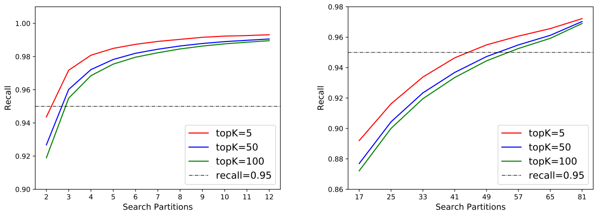
Figure 10: Recall comparison between VGPQ and IVFPQ.

that adjusting *n_subcells* does not significantly affect the construction time of `VGPQ`.

We issue queries of type $Q1$ to both indexes. Figure 10 shows the recall and response time on three datasets (`SIFT1B`, `Deep1B`, and `AliCommodity`). The recall of `IVFPQ` can be improved by increasing *n_clusters*, but the construction time will markedly rise. Under the same *n_clusters*, we can alternatively increase *n_subcells* for `VGPQ`, in which the recall will be improved without a sacrifice of construction time. Furthermore, `VGPQ` consistently performs better than `IVFPQ` on various datasets. Overall, `VGPQ` is a practical method for large-scale vector analysis, in terms of accuracy, build speed and index size.

6.3 Clustering-based partition pruning

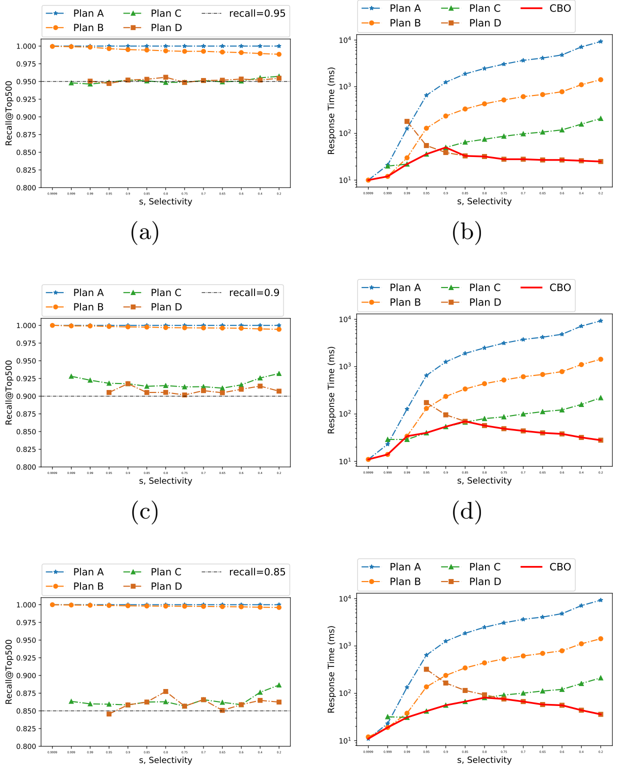
Here we demonstrate the clustering-based partition pruning effect on the query throughput in ADBV. Two distributed data tables with 512 clustering-based partitions are created for `SIFT1B` and `Deep1B` respectively. During the data ingestion, records are distributed based on the similarity to the centroids of respective clusters. When querying these tables, we can specify the number of most relevant partitions (i.e., having closest centroids to the query vector) to trade accuracy for search efficiency.



(a) SIFT1B_512p (b) Deep1B_512p

Figure 11: Performance analysis for clustering-based partition pruning.

As shown in Figure 11, the recall with respect to different top-k settings will improve as the number of searched partitions increases. However, more partitions to search also means lower query throughput. We observe that this pruning is more effective for queries with relatively small k. With the help of clustering-based partition pruning, we can reduce the number of searched partitions from 512 to 3 without violating 95% recall on `SIFT1B_512p` (Figure 11(a)). In this case, the total QPS can be improved by 100+ times, as multiple queries can be concurrently handled by different partitions (*i.e.*, read nodes). Likewise, QPS on `Deep1B_512p` also improves by 10+ times ideally. According to our empirical experience, clustering-based partition pruning helps ADBV to achieve orders of magnitude throughput improvement (*i.e.*, 10× to 100×) on large-scale datasets (with 1000+ storage nodes) for queries with relatively small k, which is common in real applications.



(a) (b)

(c) (d)

(e) (f)

Figure 12: Performance study of different physical plans.

6.4 Hybrid query optimization

In this section, we evaluate the accuracy-aware cost-based optimization for *hybrid query* execution (§5). We demonstrate that the proposed approach is able to guarantee the accuracy of query results while find the optimal physical plan in a large variety of scenarios. Since two public datasets do not contain structured columns, we only conduct following tests on `AliCommodity`. In addition, we only sample one percent tuples from the original dataset (about 8 million tuples) and construct a single-partition table. We force ADBV to perform $Q2$ (*i.e.*, *hybrid query*) with four different physical plans, for each of which we collect the average execution time and recall respectively.

In business scenarios, users have diverse requirements on the value of k and query accuracy. For example, the value of k in the face recognition scene is fairly small (*i.e.*, only a few results are required), but the accuracy is stringent (*e.g.*, *recall* > 0.99). However, in some other scenes like the E-commerce example presented in Figure 1, larger k is needed (*e.g.*, several hundreds), and the accuracy requirement is relaxed (*e.g.*, *recall* > 0.9). In order to cover divergent scenarios, we select three representative settings in this experiment:

(a) $k = 50$, *recall* $>= 0.95$, and s varies from 0.2 to 0.9999; results are showed in Figure 12(a) and 12(b).

(b) $k = 250$, *recall* $>= 0.9$, and s varies from 0.2 to 0.9999; results are showed in Figure 12(c) and 12(d).

(c) $k = 500$, *recall* $>= 0.85$, and s varies from 0.2 to 0.9999; results are showed in Figure 12(e) and 12(f).

As illustrated in Figure 12 (left column), most of these plans can provide results satisfying the recall under different selectivity. It implies that our accuracy-aware cost-based optimizer(`CBO`) is able to find proper hyperparameters for each physical plan. Meanwhile, in all three representative cases, the optimizer can always choose the optimal plan among four physical plans, as shown in Figure 12 (right column). It confirms that proposed cost models are helpful in most scenarios.

6.5 Two-step solution vs. ADBV

In this section, we compare the hybrid query processing capability of ADBV with existing two-step solutions. To the best of our knowledge, there is no publicly available system that have native support for *hybrid query* so far. Therefore, we compare with a solution following the standard practice in the industry. It combines a relational database for structured data and an `ANNS` engine for unstructured data to obtain tuples of interest. Two systems run separately and the intersection of results collected from both systems are processed afterward. To make a fair comparison, we first use AnalyticDB to retrieve tuples that satisfy the structured filter conditions. Then, nearest neighbors are computed with different `ANNS` methods (*i.e.*, `IVFPQ` and `VGPQ`). Finally, we merge result sets from above two steps to collect the tuples that pass both structured and unstructured filter conditions. The results of these two distinct two-step solutions are denoted as `AnalyticDB+IVFPQ` and `AnalyticDB+VGPQ` respectively. Since the data transfer time between systems is subjected to development environment, we ignore the time cost of data transfer in this test. In other words, the underlying execution process of `AnalyticDB+IVFPQ` and `AnalyticDB+VGPQ` conceptually follow `Plan D` (Section 5.1). We also implement `IVFPQ` in ADBV to verify the effectiveness of hybrid query optimization techniques (Section 5.2) to other existing `ANNS` algorithms. Here, `AnalyticDB-V(IVFPQ)`(or `AnalyticDB-V(VGPQ )`) represents the uniform solution developed with `IVFPQ` (or `VGPQ`).

We use query template $Q2$ to generate queries, where c is instantiated with column *create_time* that manifests the create time of each tuple. We fix p_1 and vary the value of p_2 in the structured predicate to generate three types of queries. These queries are to fetch top-50 records similar to the query image within last *one* month($p_2 = 1$), *three* months($p_2 = 3$), and *nine* months ($p_2 = 9$) respectively. Experimental results are provided in Table 3.

As showed in Table 3, `VGPQ` consistently outperforms `IVFPQ` in both two-step solutions. When $p_2 = 1$ and $p_2 = 3$, we need amplify the number of nearest neighbor to be fetched to $s \times k$ ($s > 1$) instead. The reason is that tuples are rejected by the corresponding selective structured predicates. Once p_2 is set to 9, more tuples could pass the time constraint, and thus the running time is reduced by decreasing the value of s. For highly selective queries (*e.g.* $p_2 = 1$ and $p_2 = 3$), ADBV picks `Plan A` or `Plan B` for execution, while `Plan D` is chosen until $p_2 = 9$. The differences in running time between ADBV and two-step solutions also indicate that hybrid query optimization techniques enable ADBV to automatically select the optimal execution plan for different queries.

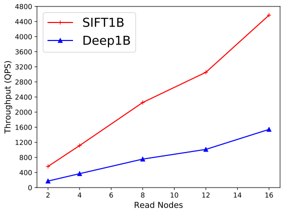


Figure 13: Scalability experiment with different numer of read nodes.

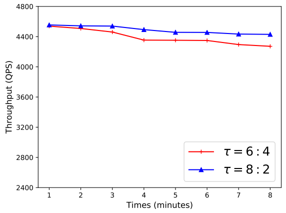


Figure 14: Performance study on mixed read/write workloads.

Table 3: Two-step solution vs. ADBV

	$p_2 = 1$	$p_2 = 3$	$p_2 = 9$
`AnalyticDB+IVFPQ`	241 ms	77 ms	47 ms
`AnalyticDB+VGPQ`	181 ms	55 ms	33 ms
`AnalyticDB-V(IVFPQ)`	19 ms	35 ms	47 ms
`AnalyticDB-V(VGPQ)`	19 ms	35 ms	33 ms

6.6 Scalability and mixed read & write

Scalability. We evaluate the peak QPS in vary-sized clusters (with 4, 8, 12, and 16 nodes) for `SIFT1B` and `Deep1B`. As shown in Figure 13, QPS increases linearly with the number of nodes.

Mixed read-write throughput. This test is conducted on `SIFT1B` with 8:2 and 6:4 read-write workload ratio. We observe that ADBV achieves high throughput (˜4400 QPS) under different scenarios. As the stress test continues, Figure 14 shows that the throughput only drops mildly. It confirms that the influence of high-rate data ingestion to the query performance is marginal under our *lambda* framework.

6.7 Use case study

ADBV has been successfully deployed at Alibaba to provide hybrid query processing services for various real-world applications. As known, Smart City Transportation significantly improves citizens' daily life by reducing traffic congestion and optimizing the traffic flow [36]. There is one important system in it, called *vehicle peccancy detection system*, which helps to identify peccancy vehicles from snapshots collected by road-junction cameras.

In this large-scale production system, ADBV is adopted as a key module to support hybrid query analysis over 20,000 video streams. For each video stream, images containing vehicles are extracted through key-frame-detection [50] algorithms at the first step. Then, each image is processed via image detection algorithms [41, 31] to locate the sub-region of pixels representing vehicles, and these sub-regions are transformed into vector points with the usage of trained deep neural network models. Meanwhile, other important structured attributes, like *timestamp*, *location*, *camera id*, *vehicle* *color*, are recorded as well. Vector points along with such structured information are organized into batches. Before inserted through issuing INSERT statements (Section 2.2), each batch of data is properly cleaned. After the completion of data ingestion, practitioners could directly rely on ADBV to manage data without concerning the underlying implementation. Until now, the volume of raw data reaches about 30TB[2] and the total number of tuples is over 13 billions. The entire process is illustrated in Figure 15.

If one user decides to retrieve one-day trajectories of a black peccany vehicle ADBV in a given area with ADBV, she/he simply prompts a SQL query by providing the time and position constraints as well as one snapshot of this vehicle. Upper layers of the system collect the results returned by ADBV, and provide the trajectory information at the user interface as showed in Figure 15 (where the red line denotes the desired trajectory). By virtue of the overall system design, VGPQ and hybrid query optimization techniques, ADBV significantly reduces the total running time of such queries from hundreds of seconds to milliseconds to satisfy the real-time requirement.

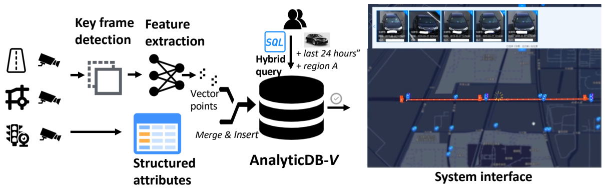
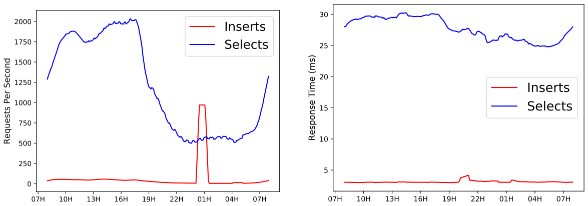


Figure 15: Use case study.

Moreover, we collect a set of 24-hour statistics from a customer's 70-node cluster to demonstrate the effectiveness and efficiency of ADBV in this production environment. As illustrated in Figure 16, ADBV consistently provides stable service either for insertions or for selections. Though the number of requests wildly fluctuates in a 24-hour time window due to business nature, the response time remains in a particular range. Even when confronting the burst of data ingestion at mid-night (*i.e.*, 1am), no noticeable performance jitters is observed. After all, not only does ADBV reduces the maintenance efforts significantly over a large-scale dataset, but also it provides reliable and efficient data analytics services.



(a) Requests per second (b) Average Response Time per operation

Figure 16: Performance in production environment.

7. RELATED WORK

ADBV is designed and implemented from the system perspective to support *hybrid queries* for structured and unstructured data with real-time updates. We survey related work in this section.

OLAP system. Most OLAP systems like OLAP databases (Vertica [28], Greenplum [1], *etc.*), batch processing systems (Spark-SQL [3]), analytical cloud services (Amazon Redshift [15], Google BigQuery [46] and Alibaba AnalyticDB [53]) have been used in a wide range of fields and provided excellent analytic ability for users in practice. However, these systems only work on traditional structured datasets and do not support *hybrid queries* containing unstructured data.

Solutions for searching unstructured data. Facebook publishes a famous vector similarity search library (also called Faiss [25]) that supports several existing ANNS algorithms. Jegou *et al.* implement the extension on Elasticsearch [9] to support ANNS over vector data. Microsoft develops GRIP [54], a vector search solution that adopts HNSW to reduce the expensive centroid searching cost of encoding-based algorithms and supports different kinds of vector indexes [23, 10]. Recently, unstructured data processing arise a lot of interests [30, 37, 57].

Nearest neighbor search. To reduce the running time of the brute force solution, several tree-based algorithms have been proposed with theoretical guarantees [16, 43]. Due to the *curse of dimensionality* [22], these algorithms are no better than exhaustive search on high dimensional features [51]. Consequently, ANNS becomes one promising direction to address this problem without loss of too much performance [22, 21, 23, 4, 33, 10]. Encoding-based approach is an alternative way to reduce the distance computation cost [13] by compressing the original vector with a succinct encoding. PQ and its subsequent works [11, 26] still proceed in an exhaustive search fashion, and the searching cost on large scale datasets are not well treated. Then, several fine-grained design algorithms are introduced, especially for immense scale datasets [4, 23]. To fulfill the high accuracy requirement, the neighborhood-based approach [33, 35, 10] is a compelling option. However, the entire index structure should be maintained in the internal memory, which will fall over when it cannot be fitted into memory.

Moreover, *hybrid query* processing is a common term which is also adopted in the information retrieval (IR) community [44, 7] as well. IR methods usually rest on statistical models to smoothly combine keyword-based search and pure semantic search into one on structured data (*e.g.*, RDF data) and unstructured data (*e.g.*, Textual documents). Such solutions for ranking tasks could not be manipulated in the design of an analytical engine so far [52].

8. CONCLUSION

In order to accommodate this strongly forecast future demand, this paper introduces a new analytic engine ADBV. With the aid of ADBV, practitioners are able to manage massive high-dimensional vectors and structured attributes by driving a single system. The proposed VGPQ algorithm could further improve the performance of hybrid query processing on a large volume of baseline data. Moreover, *hybrid query* is natively optimized by the design of accuracy-aware cost-based optimization. ADBV has been successfully deployed in Alibaba Group and on Alibaba Cloud to support various complex business scenarios. An interesting future and ongoing work is to support complex ETL processing with hybrid query semantics in the same engine for online interactive query processing.

[2] The size of raw images stored is not counted, we only calculate the quantity of vector data.

9. REFERENCES

[1] Greenplum. https://greenplum.org/.

[2] Pangu. https://www.alibabacloud.com/blog/pangu—the-high-performance-distributed-file-system-by-alibaba-cloud 594059.

[3] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al. Spark sql: Relational data processing in spark. In *SIGMOD*, pages 1383–1394. ACM, 2015.

[4] A. Babenko and V. Lempitsky. The inverted multi-index. *IEEE transactions on pattern analysis and machine intelligence*, 37(6):1247–1260, 2014.

[5] A. Babenko and V. Lempitsky. Efficient indexing of billion-scale datasets of deep descriptors. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2055–2063, 2016.

[6] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[7] R. Bhagdev, S. Chapman, F. Ciravegna, V. Lanfranchi, and D. Petrelli. Hybrid search: Effectively combining keywords and semantic searches. In *European semantic web conference*, pages 554–568. Springer, 2008.

[8] M. Douze, A. Sablayrolles, and H. Jégou. Link and code: Fast indexing with graphs and compact regression codes. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3646–3654, 2018.

[9] Elasticsearch. Elasticsearch Approximate Nearest Neighbor plugin, Jan. 2019.

[10] C. Fu, C. Xiang, C. Wang, and D. Cai. Fast approximate nearest neighbor search with the navigating spreading-out graph. *PVLDB*, 12(5):461–474, 2019.

[11] T. Ge, K. He, Q. Ke, and J. Sun. Optimized product quantization for approximate nearest neighbor search. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2946–2953, 2013.

[12] A. Gionis, P. Indyk, R. Motwani, et al. Similarity search in high dimensions via hashing. In *PVLDB*, volume 99, pages 518–529, 1999.

[13] R. Gray. Vector quantization. *IEEE Assp Magazine*, 1(2):4–29, 1984.

[14] P. J. Grother, M. L. Ngan, and K. K. Hanaoka. Face recognition vendor test (frvt) part 2: Identification. Technical report, 2019.

[15] A. Gupta, D. Agarwal, D. Tan, J. Kulesza, R. Pathak, S. Stefani, and V. Srinivasan. Amazon redshift and the case for simpler data warehouses. In *SIGMOD*, pages 1917–1923. ACM, 2015.

[16] A. Guttman. *R-trees: A dynamic index structure for spatial searching*, volume 14. ACM, 1984.

[17] J. A. Hartigan and M. A. Wong. Algorithm as 136: A k-means clustering algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 28(1):100–108, 1979.

[18] R. He, Y. Cai, T. Tan, and L. Davis. Learning predictable binary codes for face indexing. *Pattern recognition*, 48(10):3160–3168, 2015.

[19] I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Computing Surveys (CSUR)*, 40(4):11, 2008.

[20] I. F. Ilyas, R. Shah, W. G. Aref, J. S. Vitter, and A. K. Elmagarmid. Rank-aware query optimization. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 203–214. ACM, 2004.

[21] P. Indyk. Approximate nearest neighbor algorithms for fréchet distance via product metrics. In *Symposium on Computational Geometry*, pages 102–106. Citeseer, 2002.

[22] P. Indyk and R. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 604–613. ACM, 1998.

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

[24] H. Jégou, R. Tavenard, M. Douze, and L. Amsaleg. Searching in one billion vectors: re-rank with source coding. In *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 861–864. IEEE, 2011.

[25] J. Johnson, M. Douze, and H. Jégou. Billion-scale similarity search with gpus. *IEEE Transactions on Big Data*, 2019.

[26] Y. Kalantidis and Y. Avrithis. Locally optimized product quantization for approximate nearest neighbor search. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2321–2328, 2014.

[27] J. M. Kleinberg. Navigation in a small world. *Nature*, 406(6798):845, 2000.

[28] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandiver, L. Doshi, and C. Bear. The vertica analytic database: C-store 7 years later. *PVLDB*, 5(12):1790–1801, 2012.

[29] C. Li, K. C.-C. Chang, I. F. Ilyas, and S. Song. Ranksql: query algebra and optimization for relational top-k queries. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 131–142. ACM, 2005.

[30] J. Li, H. Liu, C. Gui, J. Chen, Z. Ni, N. Wang, and Y. Chen. The design and implementation of a real time visual search system on jd e-commerce platform. In *Proceedings of the 19th International Middleware Conference Industry*, Middleware '18, page 9–16, New York, NY, USA, 2018. Association for Computing Machinery.

[31] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg. Ssd: Single shot multibox detector. In *European conference on computer vision*, pages 21–37. Springer, 2016.

[32] X. Liu, W. Liu, H. Ma, and H. Fu. Large-scale vehicle re-identification in urban surveillance videos. In *2016 IEEE International Conference on Multimedia and Expo (ICME)*, pages 1–6. IEEE, 2016.

[33] Y. Malkov, A. Ponomarenko, A. Logvinov, and V. Krylov. Approximate nearest neighbor algorithm based on navigable small world graphs. *Information Systems*, 45:61–68, 2014.

[34] Y. A. Malkov and D. A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence*, 2018.

[35] Y. A. Malkov and D. A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence*, 2018.

[36] T. Michelle and E. Leonie. Alibaba's 'City Brain' is slashing congestion in its hometown . https://edition.cnn.com/2019/01/15/tech/alibaba-city-brain-hangzhou/index.html/, 2019. [Online; accessed 2-March-2020].

[37] C. Mu, J. Zhao, G. Yang, J. Zhang, and Z. Yan. Towards practical visual search engine within elasticsearch. *arXiv preprint arXiv:1806.08896*, 2018.

[38] S. M. Omohundro. *Five balltree construction algorithms*. International Computer Science Institute Berkeley, 1989.

[39] B. J. Oommen and L. G. Rueda. The efficiency of histogram-like techniques for database query optimization. *The Computer Journal*, 45:494–510, 2002.

[40] G. Piatetsky-Shapiro and C. Connell. Accurate estimation of the number of tuples satisfying a condition. *ACM Sigmod Record*, 14:256–276, 1984.

[41] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.

[42] F. Richardson, D. Reynolds, and N. Dehak. A unified deep neural network for speaker and language recognition. *arXiv preprint arXiv:1504.00923*, 2015.

[43] J. T. Robinson. The kdb-tree: a search structure for large multidimensional dynamic indexes. In *Proceedings of the 1981 ACM SIGMOD international conference on Management of data*, pages 10–18. ACM, 1981.

[44] C. Rocha, D. Schwabe, and M. P. Aragao. A hybrid approach for searching in the semantic web. In *Proceedings of the 13th international conference on World Wide Web*, pages 374–383, 2004.

[45] Y. Rui, T. S. Huang, and S.-F. Chang. Image retrieval: Current techniques, promising directions, and open issues. *Journal of visual communication and image representation*, 10:39–62, 1999.

[46] K. Sato. An inside look at google bigquery.(2012). *Retrieved Jan*, 29:2018, 2012.

[47] F. Schroff, D. Kalenichenko, and J. Philbin. Facenet: A unified embedding for face recognition and clustering. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 815–823, 2015.

[48] M. Stonebraker and L. A. Rowe. The design of postgres. *ACM Sigmod Record*, 15:340–355, 1986.

[49] J. Suchal and P. Návrat. Full text search engine as scalable k-nearest neighbor recommendation system. In *IFIP International Conference on Artificial Intelligence in Theory and Practice*, pages 165–173. Springer, 2010.

[50] C. Sujatha and U. Mudenagudi. A study on keyframe extraction methods for video summary. In *2011 International Conference on Computational Intelligence and Communication Networks*, pages 73–77. IEEE, 2011.

[51] R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *PVLDB*, volume 98, pages 194–205, 1998.

[52] G. Weikum. Db&ir: both sides now. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 25–30, 2007.

[53] C. Zhan, M. Su, C. Wei, X. Peng, L. Lin, S. Wang, Z. Chen, F. Li, Y. Pan, F. Zheng, et al. Analyticdb: Real-time olap database system at alibaba cloud. *PVLDB*, 12(12), 2019.

[54] M. Zhang and Y. He. Grip: Multi-store capacity-optimized high-performance nearest neighbor search for vector search engine. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, pages 1673–1682, 2019.

[55] Z. Zhang, C. Li, Y. Tao, R. Yang, H. Tang, and J. Xu. Fuxi: a fault-tolerant resource management and job scheduling system at internet scale. *PVLDB*, 7(13):1393–1404, 2014.

[56] L. Zheng, L. Shen, L. Tian, S. Wang, J. Wang, and Q. Tian. Scalable person re-identification: A benchmark. In *Proceedings of the IEEE international conference on computer vision*, pages 1116–1124, 2015.

[57] ZILLIZ. Milvus: an open source vector similarity search engine, Oct. 2019.