team12_final_project_report

Team Members

107070024 王柏智

108012007 林明杉

108032043 張睿云

x1120115 Niranjan Acharya

Approach to implement final project

We went through

`PASE: PostgreSQL Ultra-High-Dimensional Approximate Nearest Neighbor Search Extension`

and found the following candidate algorithms:

1. Tree based algorithm

   Effective for low-dimensional vectors, suffers significant performance degradation among high-dimensional data space.

2. Quantization based algorithm — HNSW

   Groups the vector data into clusters to avoid brute-force search.

   This method own both simplicity and high accuracy. But not suitable for very large datasets.

3. Graph-Based algorithm — IVFFlat

   Finds similar vectors through greedily scanning through neighbors.

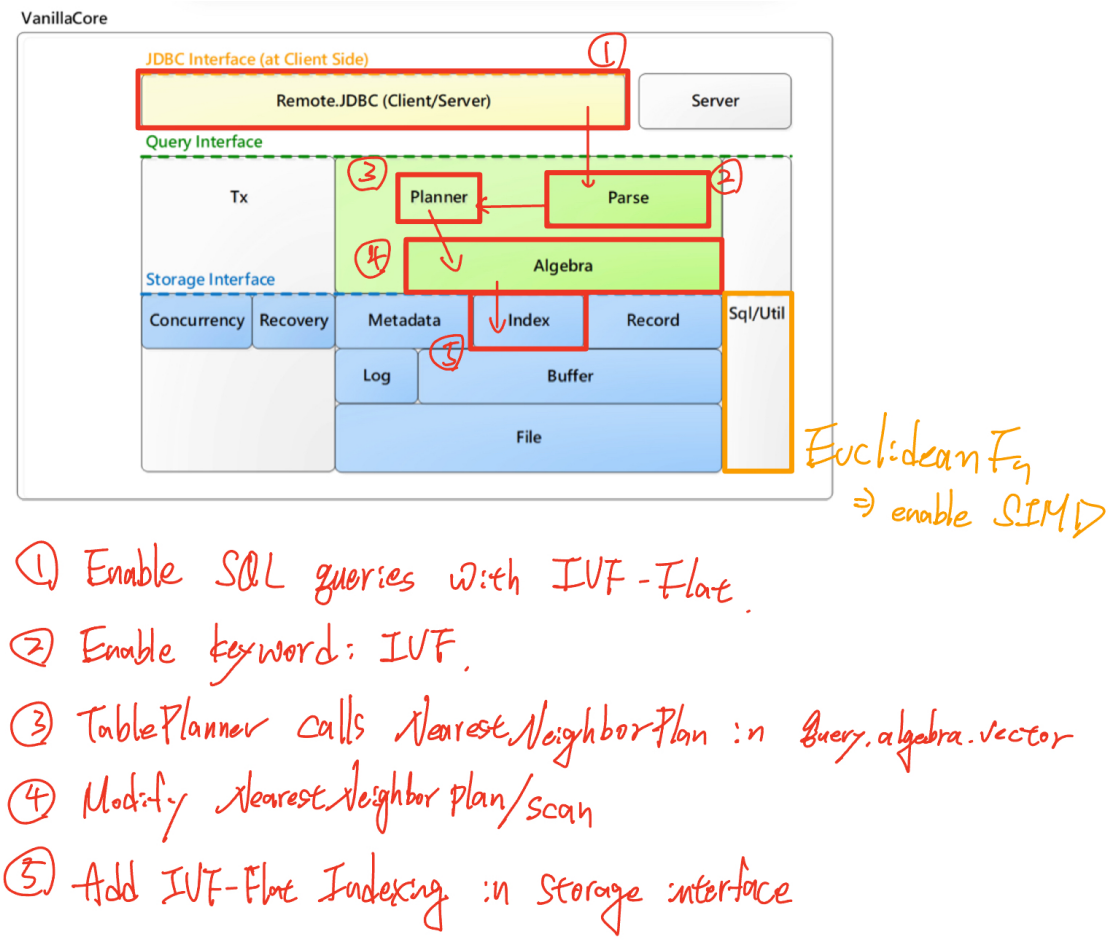
   This algorithm performs well on large datasets, but it requires a relative long time on building the neighbor graph.

4. Hash-Based algorithm

   Simple implementation, lower accuracy in comparison with other algorithms

Since we had about 30 minutes to build the neighbor graph, we finally chose IVFFlat as our implementation method.

Index Implementation procedure



1. Enable SQL queries with IVFFlat

This part is given by the template offered

2. Enable keyword IVF

In `vanilladb.core.query.parse.Lexer` add in `ivf` into the keyword list.

Enable `vanilladb.core.query.parse.Parser` to match and eat keyword: `ivf`

3. TablePlanner calls NearestNeighborPlan

In `core.query.planner.opt.TablePlanner.makeSelectPlan` NearestNeighborPlan is called to start planning the method for vectors.

This function acts as a bridge to connect (4)&(5) to (1)&(2) noted in the figure given above.

4. Modify NearestNeighborPlan NearestNeighborScan

In `core.query.algebra.vector`, both `NearestNeighborPlan` and `NearestNeighborScan` have been modified to facilitate cluster calculations. This meets the requirements for finding the kMeans.

- `NearestNeighborPlan`

  The `NearestNeighborPlan` class implements a query plan for performing nearest neighbor searches using an index structure.

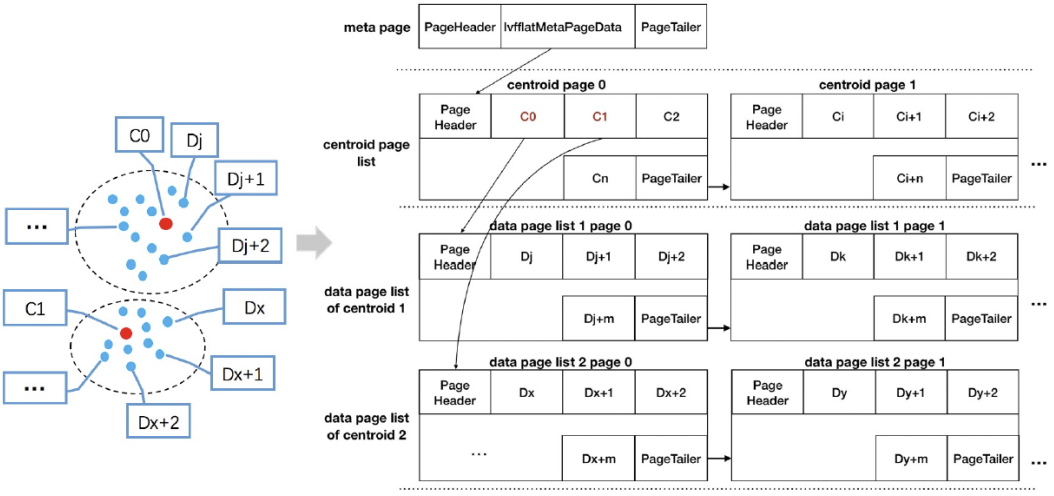
- `NearestNeighborScan`

  The `NearestNeighborScan` class implements a scan for performing nearest neighbor searches.

- `NearestNeighborUtils.PriorityQueueScan`

  The `PriorityQueueScan` class implements a scan using a priority queue for managing nearest neighbor results.

These classes collectively implement the functionality for performing nearest neighbor searches in a database using index structures and priority queues to efficiently manage and retrieve the closest records based on a distance function.



According to this structure, we use the priority queue to organize the centroid, this will make the search faster.

5. Add IVFFlat indexing in storage interface

`IvfflatIndex`

The `IvfflatIndex` class implements an index structure using IVFFlat clustering techniques for efficient similarity search.

`IvfflatIndexTrainer`

The `IvfflatIndexTrainer` class is responsible for training the IVFFlat index.

It is performing the kMeans algorithm and tries to distribute the data into each cluster to make the index of IVFFlat.

Both classes work together to create, manage, and optimize an IVF flat index structure for efficient data retrieval in the VanillaDB database system.

However, there are still some bugs in our code that keeps us from evenly distributing the data into the clusters.

SMID implementation

Euclidean function

The `calculateDistance` function uses SIMD (Single Instruction, Multiple Data) capabilities to compute the Euclidean distance between two vectors efficiently. This parallelism significantly accelerates computation by reducing the number of iterations and leveraging wide CPU registers, resulting in faster and more efficient distance calculations compared to traditional element-wise processing.

Suppose we use `IntVector.SPECIES_256`, with an integer being 4 bytes and a VectorConstant having a dimension of d. Consequently, an IntVector created based on this species can operate on $(256/(4 \times 8)) = 8$ integer data elements simultaneously, resulting in a complexity of $O(d/8)$(assuming other overheads are negligible). When creating an IntVector, an integer array is required, and VectorConstant can obtain a pre-stored integer array via asJavaVal(). Therefore, converting VectorConstant to an integer array does not need additional time.

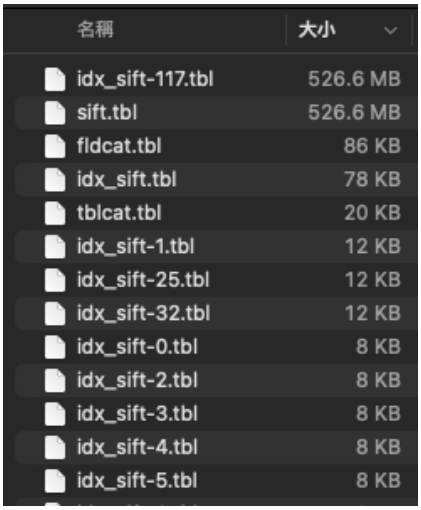
Experiment

Experiments

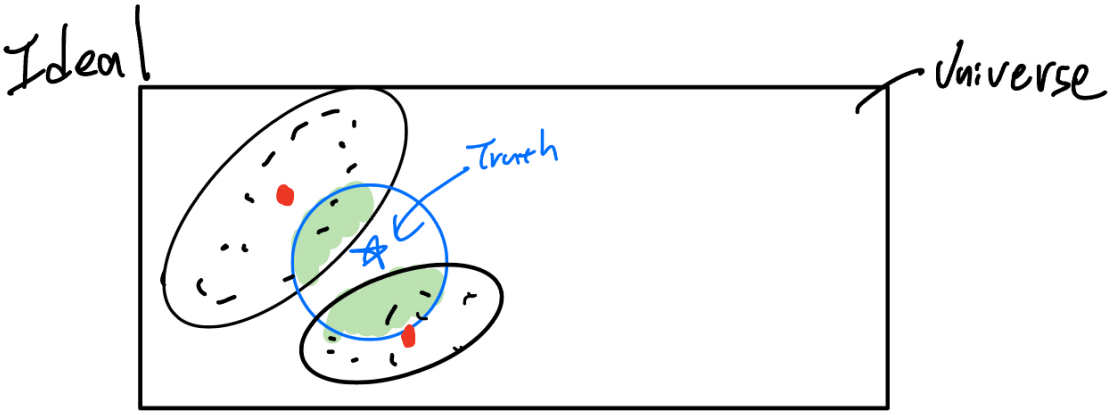
- Experiment environment:

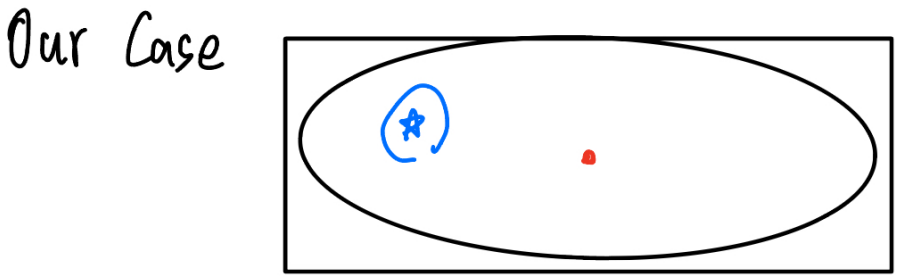
  m1 pro chip CPU @ 3.2GHz, 16GB RAM, 512GB SSD, macOS Sonoma 14.4.1

As mentioned above, we attempted to implement the IVFFlat clustering as mentioned above. However, in as the experiment ran, we found that our method cannot efficiently separate the data read into different clusters



this made our recall rate very low, as almost all of the data are placed in one cluster, picking to find the ground truth will become very hard in this data structure.





Ideally, when the cluster is small, selecting data from these small clusters makes it easy to find the true value. However, in our scenario, where most elements belong to the same cluster, it's still quite possible to select items that do not match the true value.

Experiment results

- Original performance: the given implementation did not give out transaction

Parameters:

```
org.vanilladb.bench.benchmarks.sift.SiftBenchConstants.NUM_ITEMS=9000
org.vanilladb.bench.benchmarks.sift.SiftBenchConstants.NUM_DIMENSIONS=128
org.vanilladb.bench.benchmarks.sift.SiftBenchConstants.READ_INSERT_TX_RATE
org.vanilladb.bench.benchmarks.sift.SiftBenchConstants.DATASET_FILE=sift.t
```

Since our cluster did not distribute properly, 900k benchmarking will take tens of hours to finish calculating the recall. (since the cluster file is so big, plenty of time is spent on doing I/O)

So we tried to make the dataset smaller to see if our implementation worked.

NUM_ITEMS = 9000

	Insert committed / aborted	Insert latency	ANN committed / aborted	ANN latency	Recall
Exp1	384 / 679	4	8826 / 125	12	17.27%
Exp2	375 / 592	6	8719 / 191	12	3.11%

From our experiments, it's evident that the recall rate is quite unstable. This instability might be due to the data from the larger cluster not aligning closely with the ground truth, resulting in a low and fluctuating recall rate.

NUM_ITEMS = 900k

This is still running at 2024/06/16 23:07

we will add another version of report with the results of this experiment when the experiment is finished.

（可以的話希望助教不要算遲交..）

Future works

Limited by the deadline, we did not successfully implement a feasible solution to this project. However, we did made some concrete progress in this project. Also i’m very curious where did I made the mistake that failed the data distribution.

After the final exam, I’m planning to comeback to this project to see what caused the clustering algorithm failed.