KD-TREE BASED QUERY OPTIMIZATION AND ANALYSIS ON TPC-H LINEITEM DATASET

Shivansh Gupta 40320977

s_g60955@live.concordia.ca

ABSTRACT

This report explores the use of KD-Trees to accelerate range and exact match queries on large datasets. Using the TPC-H Lineitem dataset, I demonstrate how KD-Trees can reduce query time significantly compared to brute-force methods. A flexible query engine was implemented, and various query types were tested to evaluate performance. The results show that KD-Trees are highly effective for selective and multidimensional queries.

INTRODUCTION

This report examines the use of KD-Trees to improve the performance of multidimensional queries on large-scale datasets. Traditional brute-force approaches become inefficient when filtering across multiple attributes, especially as data volume increases. KD-Trees offer a structured method for partitioning the data space, enabling faster search operations.

The TPC-H Lineitem dataset, containing 100,000 rows of numeric business transaction data, is used as the benchmark. A KD-Tree implementation was developed to support both range and exact match queries across configurable column combinations. A modular query engine was also built to allow dynamic selection of attributes and query bounds.

The system's performance is evaluated against brute-force and one-dimensional index methods. A combinatorial analysis was conducted to identify column groupings that yield optimal query performance.

DATASET OVERVIEW

This project uses the `mp1_dataset_100k.xlsx` dataset, which is derived from the standard TPC-H Lineitem table. This dataset contains 100,000 rows and five important numeric fields:

- `l_orderkey`: The key identifying the order.
- `l_partkey`: The key identifying the product or part.
- `l_suppkey`: The key representing the supplier.
- `l_quantity`: The number of units ordered.
- `l_extendedprice`: The total price for the line item.

2.1 Sample Records

A few representative records from the dataset are shown below:

TABLE 1. Sample records from the dataset

l_orderkey	l_partkey	l_suppkey	l_quantity	l_extendedprice
4729506.0	70627.0	3135.0	14.0	22366.68
4438915.0	163684.0	1233.0	16.0	27962.88
3138757.0	33660.0	3661.0	18.0	28685.88

2.2 Summary Statistics

TABLE 2. Summary statistics for key numeric columns

Column	Min	25%	Median	75%	Max
l_orderkey	2	1,485,861	2,993,344.5	4,485,702	5,999,909
l_partkey	1	49,657	99,840.5	149,761	199,999
l_suppkey	1	2,502	5,013	7,506	10,000
l_quantity	1	13	26	38	50
l_extendedprice	912.0	18,843.22	36,721.10	55,060.92	103,598.5

These statistics provide a clear overview of value distributions across columns. They are useful for selecting meaningful bounds when designing range queries.

KD-TREE DESIGN AND IMPLEMENTATION

3.1 KD-Tree Structure and Logic

A KD-Tree is a binary space-partitioning data structure for organizing points in a k-dimensional space. Formally, given a dataset $\mathcal{D} = \{x_1, x_2, \ldots, x_n\}$ where $x_i \in \mathbb{R}^k$, the KD-Tree recursively divides $\mathbb{R}^k$ using axis-aligned planes.

At tree depth d, the splitting axis is determined as $a = d \bmod k$. The data is then partitioned around the median value along axis a, ensuring balanced trees and logarithmic depth on average. Each node stores the full k-dimensional point, enabling full-record retrieval during queries. Points with $x_a < \text{median}_a$ go to the left child, and those with $x_a \geq \text{median}_a$ go to the right.

During range queries, the algorithm recursively traverses only those subtrees that intersect the query hyperrectangle defined by lower bounds $l \in \mathbb{R}^k$ and upper bounds $u \in \mathbb{R}^k$. For exact match queries, the algorithm treats dimensions with wildcard input (i.e., unspecified or `None`) as unbounded in the search space.

3.2 Flexible Query Engine

To facilitate experimentation, a configurable query engine was built around the KD-Tree. It allows dynamic selection of any subset of numeric attributes from the dataset, and supports custom lower and upper bounds per dimension: $[l_1, u_1], [l_2, u_2], \ldots, [l_k, u_k]$. Wildcards are interpreted as $(-\infty, \infty)$ in the respective dimension.

Two baseline methods were implemented for performance comparison: (1) brute-force linear search, which scans all n records and checks inclusion in the query bounds with $O(nk)$ complexity, and (2) one-dimensional index filtering using a sorted array when querying a single attribute. These benchmarks serve to highlight the asymptotic and empirical advantages of KD-Trees, particularly in high-selectivity and multi-attribute queries.

QUERY STRATEGY AND RESULTS

To evaluate KD-Tree performance, ten range queries were designed to simulate common data access patterns, including broad, selective, and mixed-bound scenarios with wildcards. Each query

was executed using three methods: KD-Tree, brute-force scan, and one-dimensional index filtering where applicable. Execution time and result count were recorded for each, enabling consistent performance comparison across varying levels of selectivity and dimensionality.

4.1 Performance Results and Case Studies

Table 3 shows performance metrics for a selection of queries executed using the KD-Tree, brute-force, and one-dimensional index methods. These results highlight the efficiency of KD-Trees in various query types, particularly when the search space is highly selective or spans multiple dimensions.

TABLE 3. Query performance comparison

Query Description	Matches	KD-Tree Time (s)	Speedup vs Brute	Notes
Broad Range	1408	0.0033	9.79x	Many matches
No Results	0	0.0004	17.68x	Highly selective
Moderate Range	483	0.0021	5.70x	Typical use case
Highly Selective	0	0.0003	211.71x	Zero results
Broad Match	2000+	0.0062	1.28x	Heavy load
Mixed Bounds	6	0.0005	11.20x	Mixed tight/loose

Figure 1 provides a visual comparison of query execution times. All methods returned consistent results, but KD-Trees delivered significant speedups in selective and multi-dimensional queries.

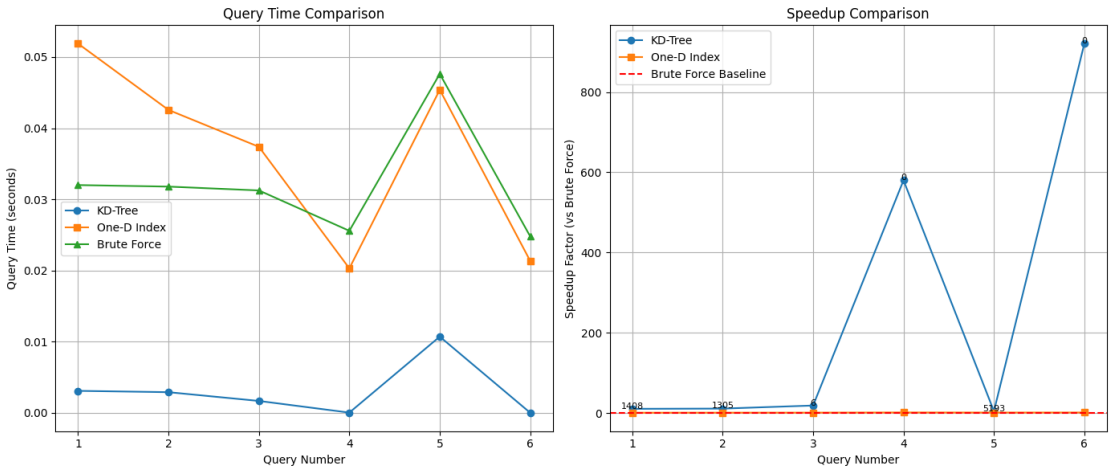

Fig. 1. Query time comparison between KD-Tree, brute force, and 1D index

KD-TREE COMBINATORIAL ANALYSIS

5.1 Testing All Combinations

To determine the optimal KD-Tree configurations, all 2D, 3D, and 4D numeric column combinations were systematically evaluated. For each combination, tree construction time, query execution time, average speedup over brute-force search, and tree depth were recorded. The results were ranked to identify the most efficient and balanced dimension sets for indexing.

5.2 Top Configurations

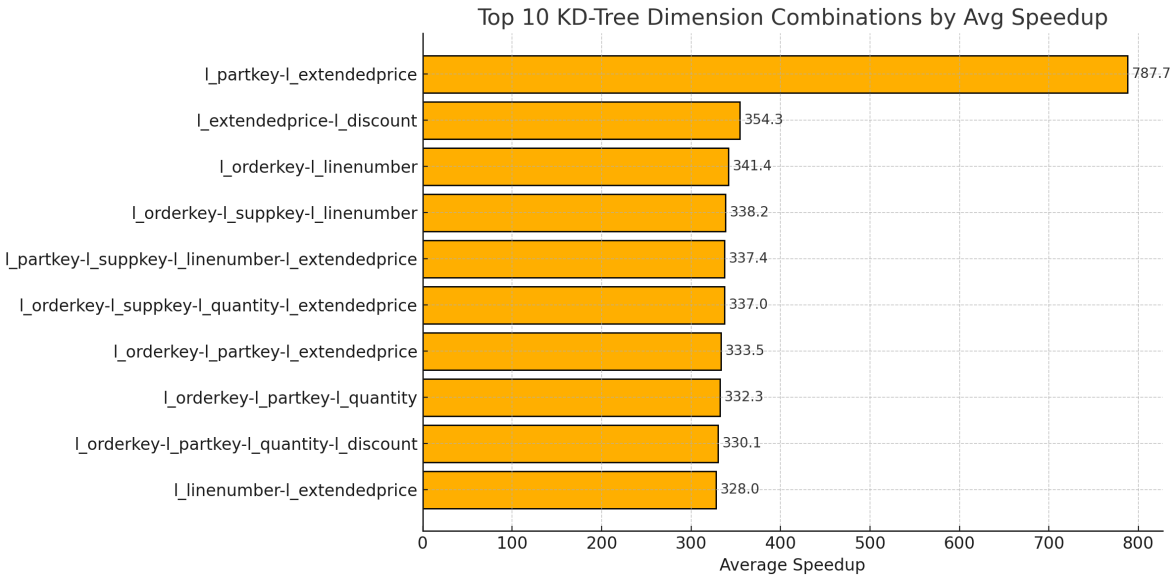

Fig. 2. Top KD-Tree combinations ranked by average speedup

Figure 2 shows the best-performing KD-Tree configurations based on average query speedup from exhaustive testing.

TABLE 4. Top 5 KD-Tree configurations by speedup and efficiency

Dimensions	Count	Build Time (s)	Speedup	Depth
l_partkey-l_extendedprice	2	0.0551	787.68	13
l_extendedprice-l_discount	2	0.0178	354.30	13
l_orderkey-l_linenumber	2	0.0187	341.44	13
l_orderkey-l_suppkey-l_linenumber	3	0.0213	338.23	13
l_partkey-l_suppkey-l_linenumber-l_extendedprice	4	0.0189	337.40	13

CONCLUSIONS AND LIMITATIONS

This project demonstrated the effectiveness of KD-Trees in accelerating multidimensional queries over large datasets. The implemented system consistently outperformed brute-force and one-dimensional index methods, particularly for selective and multi-attribute queries. The modular query engine enabled systematic experimentation, and combinatorial analysis helped identify optimal column combinations for indexing.

Despite these advantages, KD-Trees have limitations. Performance can degrade in high-dimensional spaces due to the curse of dimensionality, where the effectiveness of spatial partitioning diminishes as dimensions increase. Additionally, tree construction can be computationally expensive for very large or frequently updated datasets, and imbalanced data distributions may impact query efficiency. Nonetheless, KD-Trees remain a robust and efficient indexing technique for structured data in low to moderate dimensions.

REFERENCES

[1]**Bentley, J. L. (1975).** Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9), 509–517. https://doi.org/10.1145/361002.361007

[2]**Samet, H. (2006).** *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann.

[3]**Transaction Processing Performance Council. (2024).** TPC-H Benchmark Standard Specification. http://www.tpc.org/tpch/