Data Layout Recommendation for Big Data Systems via Large Language Models

Justin So

A thesis submitted to the

Faculty of Graduate Studies in partial

fulfillment of the requirements for the degree of

Master of Science

in

Graduate Program in Computer Science

York University

Toronto, Ontario

August 2025

© Justin So, 2025

Abstract

The physical layout of data is critical to the performance of analytical queries, especially in column-store systems like IBM Db2. Among layout strategies, Z-ordering is a popular technique that maps multi-dimensional data to a one-dimensional space while preserving locality. However, tuning Z-order is challenging: users must manually select the columns to include, and most systems assign equal weight to each column, ignoring the varying impact of different columns on query performance.

We present LayZ, an LLM-directed advisor for automated data layout tuning in IBM Db2. LayZ analyzes SQL workloads to extract query execution plan features and creates compact prompts that preserve layout-relevant information, thereby reducing inference cost when using large language models. LayZ generates ranked layout configurations, including weighted Z-orderings that adapt bit allocations based on workload characteristics. These configurations are evaluated using a cost model to identify the best candidate layout for the target workload.

Our system supports both base tables and materialized views, enabling performance recovery in queries that regress under global physical design. Experimental results on the DSB workload show that LayZ outperforms heuristic and existing layout strategies, improving query performance by up to 90%.

Dedication

I dedicate this thesis to my family, whose constant support and encouragement have been the foundation of my academic journey.

I would like to express my appreciation to my supervisors, Dr. Jarek Szlichta and Dr. Parke Godfrey. Dr. Szlichta's guidance and support have shaped a lot of my academic and personal growth, and his mentorship, along with Dr. Godfrey's, has been instrumental to my success. I thank them both for passing on their love of learning to me.

I am thankful to my IBM colleagues, Vincent Corvinelli and Calisto Zuzarte, for generously sharing their expertise and deepening my understanding of data systems.

I am also extremely grateful to my partner Erin Abila, for being with me throughout this journey.

Finally, I thank my lab mates and friends, Alexander Bianchi, Andrew Chai, and Katherine Ling. I'll fondly remember the time we spent as graduate students, it has been a great pleasure to work and grow alongside you.

Acknowledgements

I would like to acknowledge Andrew Chai and Alexander Bianchi of York University for their contribution towards the `LayZ` system. Andrew and Alex contributed to the design of the weighted Z-ordering in Chapter 1.

Contents

Abstract ii

Dedication iii

Acknowledgements iv

Table of Contents v

List of Tables viii

List of Figures ix

List of Abbreviations and Symbols x

1 Introduction 1

1.1 Motivation 1

1.2 Example of Data Layouts 5

1.3 Contributions 8

2 Background 11

2.1 Query Optimization 11

2.2 Data Layouts 15

3 Related Work **18**

3.1 Knob Tuning 18

3.2 Physical Design Structures 20

3.3 Data Layouts 21

3.4 LLM-based Tuning Configurations 25

4 System Description **27**

4.1 System Overview 27

4.2 Workload Featurizer 30

4.3 Prompt Generator 32

4.4 Configuration Evaluation 32

4.5 Configuration Selection 33

5 System Demonstration **35**

5.1 Query-Aware Data Layout Recommendations 35

5.2 Joint Interaction with Materialized View (MV)s and Data Layouts . . 40

6 Experimental Evaluation **44**

6.1 Query and Workload Level Tuning 45

6.2 Large Language Model Comparison 47

6.3 Database Knob Evaluation 48

6.4 Layout Recommendation Times 50

7 Conclusion **53**

7.1 Conclusion 53

7.2 Future Work 54

References **56**

List of Tables

2.1 Example of Zonemap. 16

5.1 Per-query template breakdown of runtime in DSB Workload 38

List of Figures

1.1 Query Execution Plan of Q_{10}. 6

1.2 Example of Z-order vs Weighted Z-order when executing query Listing 1.1. Boxes highlighted in green are blocks that have been scanned. . . 7

2.1 IBM Db2 SQL and XQuery compiler process [18]. 12

4.1 Architectural overview of LayZ. 28

4.2 Example Prompt 31

5.1 Query Execution Plan for DSB Q_{69}. 37

5.2 Main Screen of the LayZ Layout Advisor. 39

6.1 Workload Level Tuning. 46

6.2 Query level tuning. 47

6.3 LLM System Comparison. 48

6.4 LLM System Comparison on DSB Q99. 49

6.5 Sortheap Knob Setting vs Execution Time. 50

6.6 Data Layout Recommendation times of LayZ vs other. 52

List of Abbreviations and Symbols

SQL: .... Structured Query Language

DDL: .... Data Definition Language

QEP: .... Query Execution Plan

MV: .... Materialized View

LLM: .... Large Language Model

QGM: .... Query Graph Model

IBM: .... International Business Machines Corporation

DBA: .... Database Administrator

HSJOIN: .... Hash Join

MSJOIN: .... Merge Join

NLJOIN: .... Nested Loop Join

TBSCAN: .... Table Scan

DSB: .... Decision Support Benchmark

Chapter 1

Introduction

1.1 Motivation

Analytical query performance in modern databases critically depends on efficient data scanning and filtering. These operations form the backbone of analytical workloads, especially within column-store database systems such as `IBM Db2`. To accelerate query execution, database systems traditionally rely on index structures, such as B-trees and bitmaps [1], [2], materialized views (MVs) [3], and data layouts. Column-store storage itself optimizes query efficiency by restricting data scans to only relevant columns, significantly reducing I/O overhead.

Materialized views further enhance query processing efficiency by pre-computing and storing query results. This allows the database to serve subsequent queries directly from these precomputed results rather than recomputing them from base data. While effective, MVs introduce complexity, requiring careful selection and maintenance due to storage overhead and consistency considerations.

Data layouts offer another powerful technique for optimization. Traditionally,

database systems sorted data lexicographically based on selected columns, referred to as sort keys. Some database systems can take advantage of the data layout to group contiguous rows of data into blocks. These data blocks are accompanied by metadata structures known as zone maps or small materialized aggregates [4], which record the minimum and maximum values for columns within each block. During query execution, zone maps enable databases to quickly retrieve relevant data blocks, thereby reducing the number of rows scanned and significantly improving scan efficiency. For example, given a query predicate $x > 10$ for column x and a data block with a zone map indicating column x has values ranging between 1 and 5, the entire data block can be skipped, as it cannot satisfy the query predicate.

A notable technique in data layouts is *Z-ordering*, which maps multi-dimensional data into a single dimension while preserving spatial locality, significantly benefiting multi-column query filters. Z-ordering achieves this by sorting by Z-values, which are calculated by interleaving bits from each column's values, effectively encoding multi-dimensional data into a single-dimensional space. Two main variants of Z-ordering exist: equal-bit allocated Z-order and weighted Z-order. Equal-bit allocated Z-order calculates the Z-value by distributing bits evenly among selected columns, treating each column as equally significant. While straightforward, this approach may not achieve optimal performance, as different columns typically have varying impacts on query performance. Weighted Z-order, on the other hand, assigns different numbers of bits to each column based on their importance or selectivity in the workload, thereby significantly improving data locality and query efficiency over the equal-bit approach.

Traditionally, database administrators (DBAs) manually created and fine-tuned indexes, MVs, and data layouts. However, manual configuration has become increas-

ingly impractical due to the vast number of potential configurations and their associated tradeoffs, such as the overhead of maintaining data structures and workload/query-specific performance gains (i.e benefits that would be useful for one workload/query but not another). This challenge is especially pronounced in cloud environments, where thousands of databases operate simultaneously under diverse and dynamically changing workloads.

Given these challenges, physical design advisors have been created [5]–[7]. These advisors employ a variety of algorithms, ranging from heuristic pruning to machine learning, to analyze a specific query workload and metadata of the database to recommend physical design structures. Although these advisors recommend optimizations for the database, very few recommend data layouts.

Recommending data layouts is difficult for several reasons. First, it requires deep knowledge of the underlying characteristics of the data and how queries interact with these characteristics. Users must analyze selectivity, cardinality, join patterns, and data skew across multiple tables to make informed layout decisions. This process takes a long time, and the optimizations that come from the understanding of these characteristics might not apply to other workloads.

Second, the search space is enormous as there are many possible sort-key combinations, ordering strategies, and physical configurations to consider. Most database systems allow users to define a sort order over one or more columns, but this requires to user to discover the best columns for sorting. Additionally, some of these sorting methods have additional tunable parameters, such as Z-order. Real-world workloads involve complex query patterns with filters and joins across multiple columns, varied access frequencies, skewed data distributions, and evolving workloads. A layout that

benefits one query or subset of queries may degrade performance for others. As a result, exploring the search space for suitable data layouts is complex.

Third, workloads are rarely static; what works well for one workload may not suit future access patterns or changes in data volume. Real-world workloads are evolving, and thus their optimal data layout might change too. A recent work exploited Bayesian optimization to find an optimal weighted Z-ordering data layout for a given query workload and dataset [8].

Finally, the above challenges are further complicated by interactions with other performance-enhancing physical design structures such as MVs. A layout that accelerates base table scans might not be optimal for queries that utilize MVs, since the MVs can replace some parts of the query, making the data layout useless for that group of data, or data layouts can be applied on MVs, making coordinated tuning even more complex.

Recent applications of LLMs have shown success in database system tuning [9], [10]; however, LLMs, to our knowledge, have not yet been explored for data layouts. Furthermore, recent works [8], [11], [12] employ machine learning to find the near-optimal tuning configurations, yet these approaches are hindered by substantial training and exploration overheads. These challenges have motivated the use of large language models (LLMs) to recommend data layouts. Similar to domain experts, models can leverage best practices and common knowledge to prune the data layout search space to the most viable options for a given context.

1.2 Example of Data Layouts

Data layouts affect the performance of query processing by modifying the query execution process once the optimizer and execution engine consider zone maps. In other words, once the optimized access plan is finalized and ready to execute, the actual execution of the access plan would benefit from the usage of zone maps. In `IBM Db2`, their version of zone maps is called synopsis tables [13], which keep track of the minimum/maximum of each column in a data block. These are internally created column-organized tables for each user-defined table and cannot be modified directly by the user. To influence the partitions in the synopsis tables, we order the data prior to the creation of the synopsis tables.

When the difference between the minimum and maximum values for a data block is large, it will satisfy more predicates, leading to that data block being scanned more. If the difference is very large, then it could be similar to scanning the base table.

We describe the effects of data layouts using a query execution plan (QEP). A QEP details the sequence of steps the database system will take to execute a query. Figure 1.1 is the QEP for Query 10 of the Decision Support Benchmark (DSB) [14]. Note the table scan (TBSCAN) operators near the leaf nodes of the QEP. These operators reduce the cardinality of the base tables significantly in the QEP. Each TBSCAN operator in this QEP has a predicate associated with it. To reduce the number of rows scanned in TBSCAN, `IBM Db2` will use the synopsis tables to reduce the amount of data that needs to be scanned by checking if a designated predicate satisfies a particular data block. Having these synopsis tables organized in a way that allows the predicates to eliminate as many blocks as possible is essential to TBSCAN performance. This denotes the importance of improving the table scan and

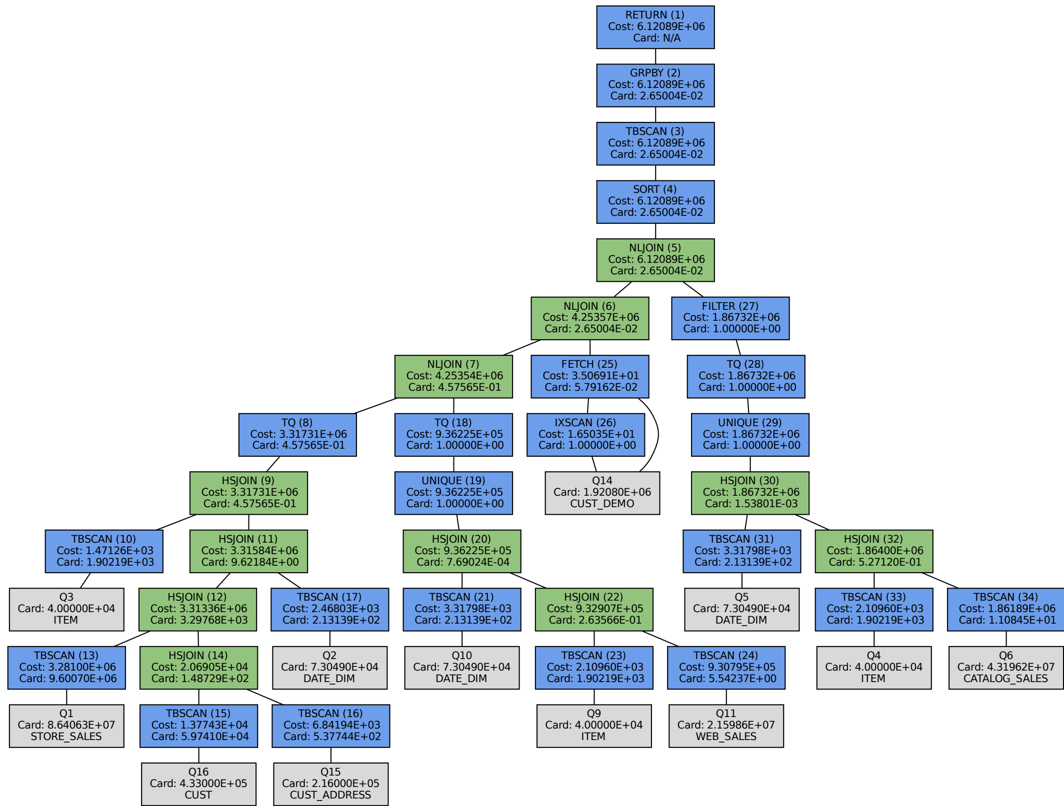
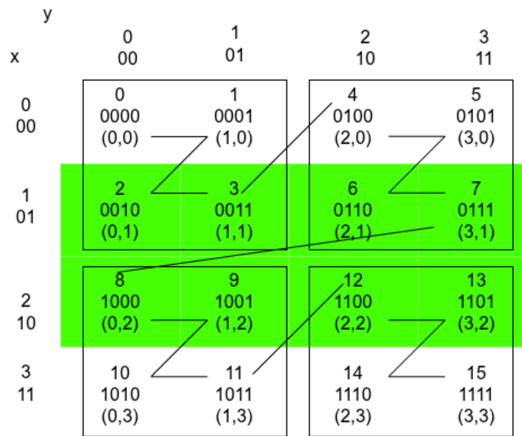


Figure 1.1: Query Execution Plan of Q_{10}.

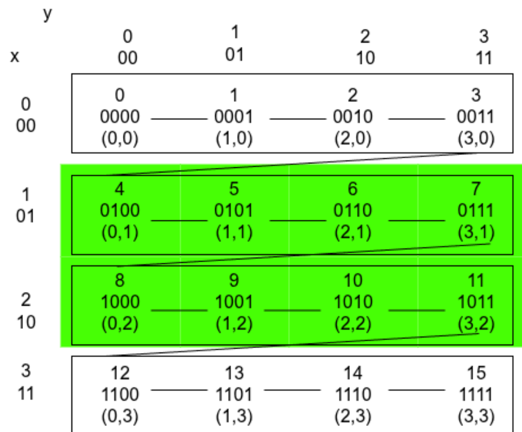
filtering performance. While we illustrate this here with only two predicates, this behavior extends to a multi-dimensional space of predicates to be tuned. Thus, this is a complex search space that is prohibitively expensive to search exhaustively and difficult to tune manually within.

Figure 1.2 illustrates how a data layout can change the number of rows scanned in a table for the following query:

```
1 SELECT * FROM store_sales
```



(a) Z-order with equal bit allocation results in 4 blocks being scanned



(b) Weighted Z-order with equal bit allocation results in 2 blocks being scanned

Figure 1.2: Example of Z-order vs Weighted Z-order when executing query Listing 1.1. Boxes highlighted in green are blocks that have been scanned.

```sql
2 WHERE x BETWEEN 0 and 3 and y BETWEEN 1 and 2
```

Listing 1.1: Example query

Each data block contains 4 rows. The query accesses 4 blocks of data under the Z-order scenario when sorting with equal bit allocation. Since the query has 2 columns in the predicate, we Z-order by those 2 columns. When the table is weighted Z-ordered by the two columns, the query only needs to access 3 blocks of data. The weighted Z-ordered table allows the database to scan fewer rows, thereby improving performance. However, on a different query, weighted-bit allocated Z-order and single-column sort key scan the same number of rows. Changing the weights according to the query in the weighted bit-allocated Z-order can improve the weighted bit-allocated Z-order to be better than a single-key column sort key. Thus, tuning your data layout to the query workload can result in beneficial query performance changes. Therefore, for effective data layouts, we need to tune our data layout to the query workload, being able to consider the associated statistics of the QEPs is of value for data layout recommendation.

1.3 Contributions

We present `LayZ`, a LLM-directed data layout advisor for `IBM Db2`, which provides query-aware layout recommendations for analytical workloads. We provide an overview on the background of query optimization and data layouts in Chapter 2 and an architectural overview in Chapter 4.1. This work makes the following contributions towards addressing the challenges outlined in Chapter 1.1.

1. **Related Works (Chapter 3).** We give a comprehensive description of recent

related works in database tuning.

2. **A LLM-Directed Layout Advisor (Chapter 4).** `LayZ` leverages a *large language model* (LLM) to provide effective and efficient data layout recommendations for diverse analytical query workloads.

   (a) **LLM Data Preprocessing For Data Layouts (Chapter 4.2).** We design a data preprocessing component that will featurize the query workload and database metadata to generate a succinct prompt for the LLM. We employ this component because LLMs are known to charge by the token [15], [16] and exhibit high GPU usage [17], so reducing the number of tokens will lower the financial and resource cost of using the LLM. Previous works using LLMs in the database field for tuning have shown success with preprocessing relevant data before giving it to the LLM [9], [10]

   (b) **Database Meta-data Driven Layout Evaluation (Chapter 4.4).** We design a cost model based on Gao's work [8] that produces an estimated query runtime for use in data layout recommendations. We leverage the synopsis tables from `IBM Db2` and a uniform sample of the database tables, allowing for effective evaluation without running expensive analytical queries for evaluation. We employ these techniques as a fast and scalable evaluation metric for large workloads, where repetitive query executions are prohibitively expensive.

3. **System Demonstration (Chapter 5).** We create an interactive application for `LayZ` showing how we can recommend data layouts for data systems to boost performance, and how its recommendations aid users in creating other physical design structures to enhance the complex analytical workload.

4. **Experimental Validation (Chapter 6).**

   (a) **Data Layout System Comparisons (Chapter 6.1).** We demonstrate the efficiency of `LayZ` on DSB, showcasing significant improvements when compared against a state-of-the-art *query-aware* layout advisor and random data layout recommendations of up to 90%. This state-of-the-art layout advisor employs Bayesian Optimization (BO) to choose the best columns and their weights for a weighted Z-order configuration [8] for a singular table. We run two experiments: first, in which a data layout configuration is set for each query; and second, in which a single data layout configuration is set for the query workload.

   (b) **Different LLMs (Chapter 6.2).** Via an evaluation suite, we evaluate the effectiveness of `LayZ` using different LLMs using our different performance metrics. We evaluate the popular LLMs based on their tuning recommendations for `IBM Db2`. This demonstrates how `LayZ` can give effective recommendations using varying LLMs.

   (c) **Different Knobs (Chapter 6.3).** We evaluate the effectiveness of layouts under different knob settings. We run an experiment to evaluate the differences of knobs on layouts: to compare `LayZ` with the default settings of knobs and `LayZ` with recommended knobs based on the query workload.

   (d) **Training times (Chapter 6.4).** We evaluate the time to get recommendations from `LayZ` and the BO method. Our experiment consists of evaluating the training times in several varieties of dataset sizes.

We conclude in Chapter 7.

Chapter 2

Background

2.1 Query Optimization

Query optimization is a feature of many relational database systems. Each query produces a QEP. Each QEP has a cost associated with it. The process of finding a correct QEP with the best cost is query optimization. The performance of a query is heavily reliant on query optimization. Query optimization relies on a component called the query optimizer to apply many of these optimizations.

For example, `IBM Db2`'s query execution process heavily relies on query optimization depicted in Figure 2.1 [18]. The query execution process starts by parsing and validating the query and turning into an internal data representation model called the query graph model (QGM) (Step 1 and 2). This stage ensures that there are no logical inconsistencies or syntax errors in the query. Using the created QGM, the QGM is transformed into a form that can be optimized more easily (Step 3 and 4). This stage involves transformations like predicate pushdown which is when the level of a predicate's application can be altered, potentially improving query performance.

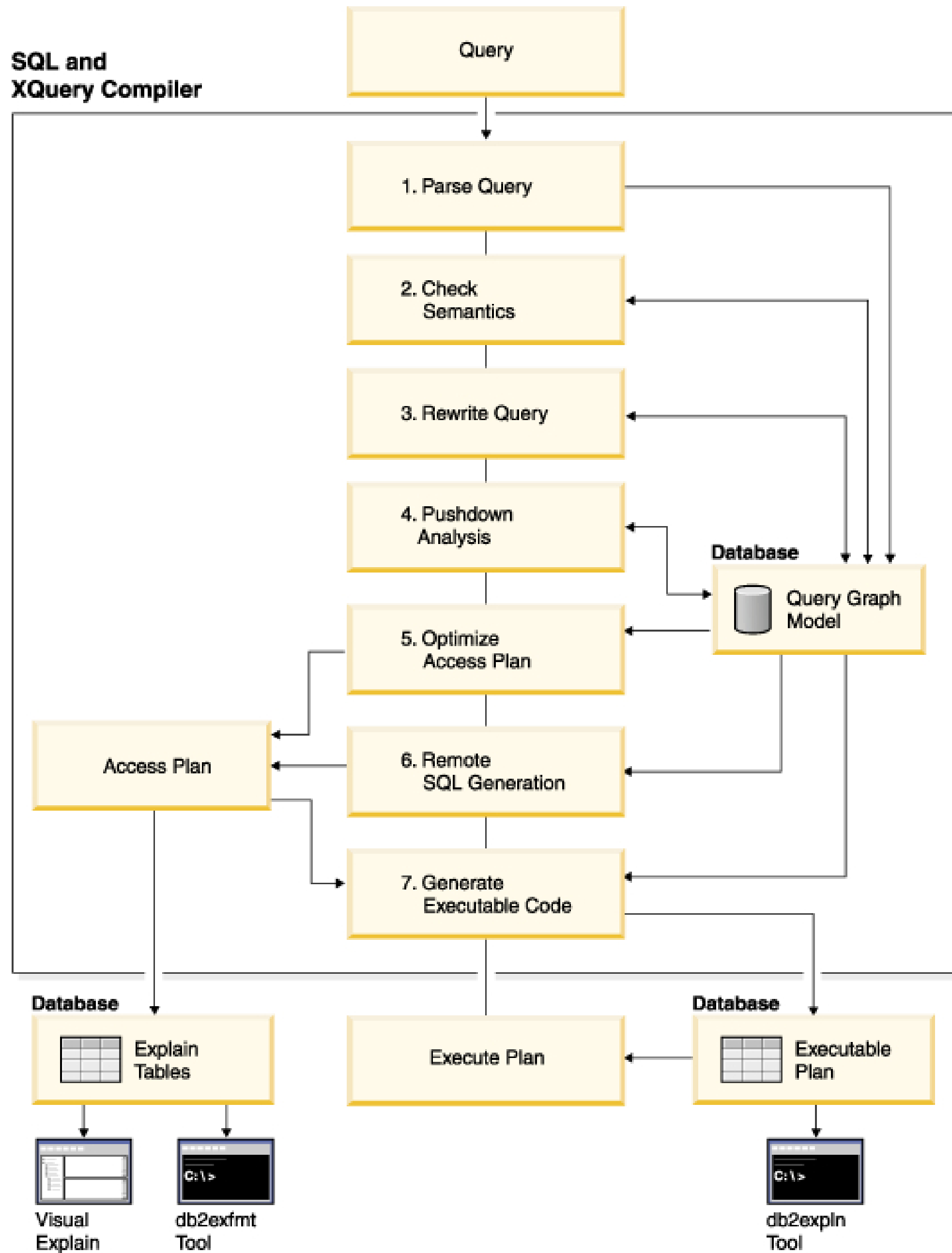


Figure 2.1: IBM Db2 SQL and XQuery compiler process [18].

After the QGM is rewritten, the QGM now goes through the query optimizer to optimize the access plan. The query optimizer is a complex component that involves generating alternative execution plans for a query, estimating the execution cost for each QEP by using metadata including table, index, column, function statistics, and `IBM Db2` registry variables [19], then choosing the plan with the smallest estimated execution cost, and outputting an access plan for the corresponding chosen execution plan (Step 6). Since the number of possible execution plans generated from a single query grows exponentially with respect to the number of tables referenced in the query, `IBM Db2` uses several different techniques to prune the search space of QEPs for efficient QEPs [20]. Some of these techniques include the algortihm used for join enumeration (greedy or dynamic programming) and which statistics to include during optimization. After the final query plan is generated, a executable query plan is created for the query. This executable access plan is then used to execute the query (Step 7).

Many relational databases have similar processes for query processing. The general process is parsing the query into an internal data model such as a logical query tree also known as an abstract syntax tree, which is then fed to the query optimizer to provide an optimized plan for the query execution engine to create an execution plan, which is a tree of physical operators with edges representing data flow between the physical operators. Query execution engine is the component that decides between physical operators such as MSJOIN and HSJOIN. The query execution engine will take in metadata to decide the best operator. Query optimization is a difficult task because the goal is to choose a correct plan with the best cost. The search space for execution plans is huge, and the optimizer must compare the costs of each of these

plans, then search for the best one among these cost estimated plans.

To illustrate why the search space is exponential with respect to the number of tables referenced in a query, consider the number of different join orderings between the tables. For example, with 3 tables, there are 12 possible join orderings. The specific formula is $\frac{(2*(N-1))!}{N-1}$ where N is the number of relations [21]. Additionally, the optimizer must consider for each logical operator in the plan, the implementation (e.g table scan versus index scan).

Cost estimation is also a difficult task. Cost estimation is used because executing all the different execution plans for the same query is infeasible, especially for more complex queries. Many databases utilize a cost model to estimate the cost of these execution plans using metadata such as table cardinality. Each operator in the execution plan will have a cost associated with it, which will be a unit of work estimated to be done by that operator during execution.

Finally, searching through the execution plans and estimating the cost of each plan exhaustively is infeasible. This is why there is a search algorithm to find the plans with the smallest estimated cost. An early example of a search algorithm is the dynamic programming approach introduced in System R [22]. The algorithm first enumerates all possible access paths for each individual relation. It then systematically combines these to evaluate all possible join orders for two relations, extends the process to three relations, and continues iteratively until all relations for the entire query are considered.

Query optimizers have to consider all these aspects to be effective in query optimization. Query optimization is an ongoing research field with a deep literature [12], [23]–[28]

2.2 Data Layouts

Data layout is one aspect that the query optimizer will consider when generating an execution plan with an optimized access plan. The importance of data layouts comes from the need to increase performance over the random access of data on non-volatile storage. Sequential access of data is a lot faster than random access of data. Therefore, it is critical for data system performance to have similar data physically co-located so queries can access the data sequentially.

Data layouts are used in analytical data systems such as `IBM Db2` with column-organized tables to improve the performance of scans and filtering over the data. Data layouts improve performance by reducing the amount of data scanned during their respective operations. One technique that these analytical data systems use is leverage data layouts that are appropriate for their query workload. For example, one can order the data by a specific column by choosing that column to be the sort key. This sort key is used to sort the data into several partitions called data blocks. Metadata, such as the minimum/maximum value for each column, is constructed for each of these blocks. This minimum/maximum value construction and storage is often called a zone map or small materialized aggregate [4], [29]. These zone maps allow the data system to skip irrelevant accesses of data blocks, thereby improving performance. For instance, Table 2.1 shows an example of a zonemap. The first two columns (SS_SOLD_DATE_SKMIN and SS_SOLD_DATE_SKMAX) show the ranges for a particular contiguous range of rows shown in the last two columns (TSNMIN and TSNMAX).

Multiple data layouts have been implemented and proposed before, such as single-column and multi-column data layouts [30]. The common data layouts are lex-

SS_SOLD_DATE_SKMIN	SS_SOLD_DATE_SKMAX	TSNMIN	TSNMAX
2451970	2452031	0	1023
2452160	2452415	1024	2047
2451456	2452474	2048	3071
2451456	2451967	3072	4095
2451776	2452223	4096	5119

Table 2.1: Example of Zonemap.

icographical sorting using single-key and compound-key columns, and Z-order for multidimensional data layouts. Lexicographical sorting uses the natural order of values to sort the values. For example, alphabetical order when dealing with letters or ascending order for numbers. Z-order involves interleaving the bits of the binary representation of the values being sorted. Z-order maps multi-dimensional values into a one-dimensional space. Z-order preserves locality of the data points, meaning that values close together in the multi-dimensional space will also be close in the one-dimensional space. Less commonly, Hilbert ordering, a technique with stronger locality preservation than Z-order, has been explored for further improving data organization in multidimensional layouts. Hilbert ordering involves recursively partitioning your data into quadrants and connecting them in a continuous path. Each point is assigned a one-dimensional Hilbert index corresponding to its position along this curve, and the points are sorted by this index value. As a result, locality preservation is stronger; however, the computation of Hilbert ordering involves many computationally intensive operations, including complex bit manipulations and recursive transformations, which make Hilbert ordering slow to compute compared to Z-order and difficult to implement in database systems.

These data layouts are implemented in many of the open table formats, such as Apache Iceberg [31]. Open table formats are becoming increasingly popular due to

the use cases they support, such as the metadata they store that helps minimize data scans and the separation of compute and storage. Popular features of these open table formats include time travel and snapshot isolation [31]. Each open table format has its tradeoffs was discussed in literature [32], [33]

Chapter 3

Related Work

3.1 Knob Tuning

Many systems for automatic database system knob tuning using machine learning have been developed over the last several years; e.g., [12], [23], [25], [34]–[38]. In this part, we focus on the ones most relevant to the `LayZ` system.

`Db2une` [23] is an automatic query-aware tuning system that leverages reinforcement learning to maximize performance while minimizing resource usage. `Db2une` employs a transformer-based query-embedding pipeline to give to the deep reinforcement learning model, which learns from cost estimates, interpolated cost estimates, and database statistics to recommend the optimal tuning configurations for a target workload. This approach does not require the system to explicitly run the queries on the database nor create a full copy of the database to run queries on, and demonstrates that cost estimates can successfully be used in knob-tuning, the system still relies on the query optimizer to give an effective cost estimates, which may limit its use with other databases where the query optimizer might be more likely to give

inaccurate cost estimates, and requires lots of training data to get effective tuning recommendations.

`OtterTune` [12], [39] uses Lasso regression to identify and rank influential knobs, matches workloads to previously collected performance data, and applies Bayesian optimization (BO) with a Gaussian process to recommend knob settings. While effective, this method requires a substantial number of costly training samples. Moreover, although BO has shown promise for knob tuning, its performance degrades in high-dimensional search spaces [40], [41], making it less suitable for complex analytical workloads.

`CDBTune` [25] introduces deep reinforcement learning (DRL) for knob tuning, leveraging Deep Deterministic Policy Gradient (DDPG), an off-policy learning algorithm. Unlike OtterTune, it learns through interaction with the database environment, reducing the reliance on large labeled datasets. However, CDBTune only supports coarse-grained tuning for broad workload types (e.g., read-heavy) and is not query-aware. It reacts solely to system-level metrics, such as data read counters and lock timeouts, without considering individual query characteristics.

Gur et al. [42] extend the DDPG approach to support workload variability via a multi-model solution, using `IBM Db2`monitoring to adapt memory allocation. Their method is well-suited to a diverse range of OLTP workloads, but training and maintaining multiple models becomes costly in environments with many distinct workload patterns.

`QTune` [34] builds upon on `CDBTune` by incorporating query features into the tuning process through a double-state DDPG (DS-DDPG) model. It featurizes queries based on involved tables, attributes, and estimated costs, and uses these features to predict

system state and recommend knob settings. However, QTune's representation fails to fully capture query structure and cardinality, limiting its ability to generalize to unseen queries and schemas.

3.2 Physical Design Structures

The problem of tuning physical design structures (PDS) has been explored for decades, with most existing tools focusing on a single type of structure [43], [44]. Index tuning has served as a natural starting point for many systems due to its relative simplicity and proven effectiveness. Numerous traditional and learned approaches have been proposed for index tuning [43], [45]–[47] Similarly, there has been substantial research into view tuning [3]. However, few tools operate in the joint design space of indexes and materialized views [5]–[7], [11]. We focus on the tools most relevant to our system.

`Db2Advisor` [6]: A comprehensive physical design recommendation tool integrated into `IBM Db2` that automatically recommends any subset of four physical design features—indexes, materialized query tables (MQTs), data partitioning, and multi-dimensional clustering (MDC)—based on a given SQL workload and storage constraints. It employs a hybrid approach that combines integrated and iterative search strategies to capture strong feature interdependencies (e.g., between indexes and MQTs) while maintaining extensibility and modularity. The system includes a workload reduction module for scalability and uses the DB2 optimizer in extended "EXPLAIN" modes to recommend and evaluate virtual design structures. These virtual design structures rely on the cost optimizer, leading to potentially bad choices if the cost optimizer is inaccurate. It recommends MDC and data partitioning, forming the early and practical instances of data layout tuning.

`HMAB` [11]: `HMAB` (Hierarchy of Multi-Armed Bandits) is a self-driving physical design tuning framework that jointly optimizes indices and materialized views using a hierarchical contextual combinatorial bandit architecture. It balances exploration and exploitation based on actual query performance, avoiding reliance on inaccurate cost models. The two-tier hierarchy first generates and filters PDS candidates (e.g., per-table indices and multi-table views), then evaluates joint configurations using contextual feedback to guide selection and learning. They leverage hypothetical what-if analysis to prune ineffective options before materialization. The design supports modular context modeling, incremental learning, and online adaptation to workload drift, making it a good foundation for including data layout recommendations.

3.3 Data Layouts

The ability to sort data along multiple dimensions has become increasingly important in modern analytical databases. To support this, many systems have adopted multi-dimensional sorting schemes, with Z-order (or Z-ordering) emerging as a popular strategy. Z-order is a locality-preserving space-filling curve that maps multi-dimensional data to a single dimension by interleaving the bits of the binary representations of each column value.

Recent systems have leveraged Z-ordering as a lightweight indexing mechanism to improve scan efficiency. Databricks Delta Lake, for example, organizes data into horizontal partitions (blocks) and maintains zone maps—metadata summaries that contain the minimum and maximum values for each block. By sorting data according to Z-order values, Delta Lake improves the effectiveness of zone maps and enhances block skipping, thereby significantly reducing the amount of data scanned during

queries [48]. Amazon Redshift [7] similarly supports Z-ordering via interleaved sort keys, which are designed to improve zone map performance. These systems often cache zone map metadata in memory, whereas standard file formats, such as Apache Parquet and Apache ORC, store such metadata in the file footer, allowing blocks to be skipped during scans if their value ranges do not intersect the query predicate.

`Delta Lake` [48]: `Delta Lake` is an open-source storage layer that brings ACID transactions to big data workloads on Apache Spark. It organizes data into horizontal partitions (blocks) and employs zone maps that record min/max statistics for each column within a block. During query execution, these statistics are used to skip irrelevant blocks whose value ranges do not intersect the query predicates. To improve the effectiveness of block skipping, `Delta Lake` sorts data using Z-ordering, which interleaves bits from multiple columns to preserve multi-dimensional locality. This sort order increases the likelihood that relevant tuples for a multi-column predicate appear within the same block, thus enhancing pruning performance and reducing scan cost. Additionally, `Delta Lake` supports Hilbert ordering through its Liquid Clustering feature [49]; however, there are no tunable parameters other than column selection for the Hilbert ordering given to the end-user.

`Amazon Redshift` [7]: Redshift is a cloud data warehouse that supports interleaved sort keys, a mechanism that approximates Z-ordering. Unlike traditional sort keys, which sort data by one column followed by others lexicographically, interleaved sort keys assign equal priority to all included columns and generate a sort order that balances their influence. This enhances the effectiveness of block-level zone maps, especially for queries with compound predicates over multiple dimensions. Like Delta Lake, Redshift relies on user-specified column selections, and poor choices can lead

to suboptimal query performance due to ineffective pruning.

`Apache Parquet and ORC` [50], [51]: These are popular open columnar file formats used in big data systems. They support lightweight metadata indexing by storing min/max statistics and other summary information in the file or block footers. During scan operations, query engines inspect this metadata to determine whether a block is relevant to the query, allowing coarse-grained block skipping. Although Parquet and ORC do not directly support Z-ordering, systems built on top of them (e.g., Delta Lake, Apache Iceberg) can enforce a Z-order-based sort order before writing data, thereby improving the selectivity of the built-in metadata filters.

`UB-tree` [52]: The UB-tree is a multi-dimensional index structure built on top of a traditional B-tree. It uses Z-order values to partition space into regions called Z-regions, where each region corresponds to an interval on the Z-order curve. Range queries are answered by computing the minimum and maximum Z-values for a rectangular query region and scanning the B-tree for all leaf pages whose intervals intersect this range. When an out-of-range Z-value is encountered, the algorithm uses geometric properties of the Z-curve to "jump ahead" to the next candidate region, thereby avoiding unnecessary scans. UB-trees enable precise filtering for range queries but incur higher storage and update costs compared to simpler approaches like zone maps.

`Amazon DynamoDB` [53], [54]: DynamoDB supports secondary indexes that can be defined using Z-order encodings of multiple attributes. For multi-dimensional range queries, it evaluates whether a given Z-order value is relevant using a function like isRelevant, and uses nextJumpIn to calculate the next minimum Z-value inside the query region. This mimics the UB-tree approach and provides better precision than metadata-only filtering. However, DynamoDB relies on external function calls to han-

dle these operations, which may add overhead during query execution. Additionally, maintaining such indexes introduces extra write amplification and storage costs.

`Flood` [55]: `Flood` is a learned, read-optimized, in-memory index designed for multi-dimensional data. It uses a workload-driven training phase that analyzes a sample of queries to identify frequent combinations of accessed columns. Based on this analysis, `Flood` generates a custom layout strategy that clusters related attributes and minimizes query access time. The index structure is adaptive, meaning it can reconfigure itself when query patterns shift, but it requires access to representative workloads and may not generalize to unseen queries without retraining.

`Tsunami` [56]: `Tsunami` extends `Flood` by incorporating techniques to handle skewed data distributions and workloads. It introduces a lightweight analytical cost model that estimates query performance for different layout configurations, allowing the system to select the most efficient one. `Tsunami` supports dynamic layout selection, enabling it to adjust to workload changes without full retraining. While more robust than `Flood`, it remains an in-memory structure focused on point and small-range queries and does not directly support persistent storage layouts such as Z-ordering with block pruning.

`Gao's Learned Z-order with Dynamic Bit Allocation` [8]: Gao proposes a learned approach to Z-order indexing that allocates a different number of bits to each column, referred to as dynamic bit allocation. This method uses Bayesian optimization to search over Z-order configurations, considering which columns to include and how many bits to assign to each column. It introduces a cost model based on query selectivity and column statistics to guide the search and uses Apache Arrow and Parquet for data storage. This reveals the value of bit-level control and query-aware

weighting, which existing systems lack.

3.4 LLM-based Tuning Configurations

LLMs have been used recently to some degree of success in the database research field. In the knob tuning field, they can recommend database-specific knobs by either reading the documentation [10] or by relying on the pre-trained weights with a compressed version of the query workload in the prompt [9]. We focus on the ones most relevant to our system.

λ-`Tune` [9] introduces a prompt-based system for automated database configuration tuning using large language models (LLMs). Unlike traditional ML-based tuning systems that rely on costly training data or system profiling, λ-`Tune` leverages the implicit knowledge encoded in instruction-tuned LLMs to recommend tuning configurations for new workloads without requiring any prior database access. The system utilizes the LLM to predict knob values and indexes based on a natural language description of the query workload, database system name, and hardware specifications. Their approach does not embed database-related statistics in the prompt for the LLM, which may limit its performance on workloads with abnormal data distribution.

`DB-BERT` [10] introduces a database tuning system that leverages large language models (BERT) and reinforcement learning to optimize system configuration parameters by extracting hints from natural language text such as manuals and tuning guides.. DB-BERT focuses on software-level tuning (e.g., buffer sizes, cost constants) rather than physical data layouts, but it illustrates a generalizable methodology: using pretrained language models to interpret tuning knowledge and combining it with workload-specific performance feedback. This paradigm of text-informed, feedback-

driven optimization is increasingly being applied to layout decisions as well, bridging configuration and physical design in learned database systems.

`GPTuner` [57] introduces a database configuration tuning system that integrates domain knowledge into the optimization process by leveraging large language models (LLMs). It constructs a structured collection of database tuning knowledge from manuals, forums, and GPT-generated insights, which is then used to narrow the search space and guide a Coarse-to-Fine Bayesian Optimization strategy. `GPTuner` selects relevant knobs without training, prunes unsafe or irrelevant value ranges, and explicitly models special-case configurations via virtual knobs.

Chapter 4

System Description

4.1 System Overview

`LayZ` is a query-aware data layout recommendation system for `IBM Db2`, designed to optimize for performance on analytical query workloads. `LayZ` leverages an LLM to generate data layout configurations for analytical query workloads. Our intuition is that the data layout recommendations can be described as a concise prompt to the LLM that already contains the necessary pre-trained weights for domain-specific and query-specific layout information. Using an LLM can lessen the need for training data and the training overheads associated with training in deep learning systems. The system's primary purpose is to make data layout recommendations on request.

Figure 4.1 overviews the architecture of `LayZ`. A *query workload* is defined as a collection of SQL queries executed over a specific database (its schema and corresponding data). A *query execution plan* (QEP) details the steps the database system undertakes to retrieve the queried data, including the chosen join order, join type, associated costs, cardinalities of intermediate results, and underlying statistics (as

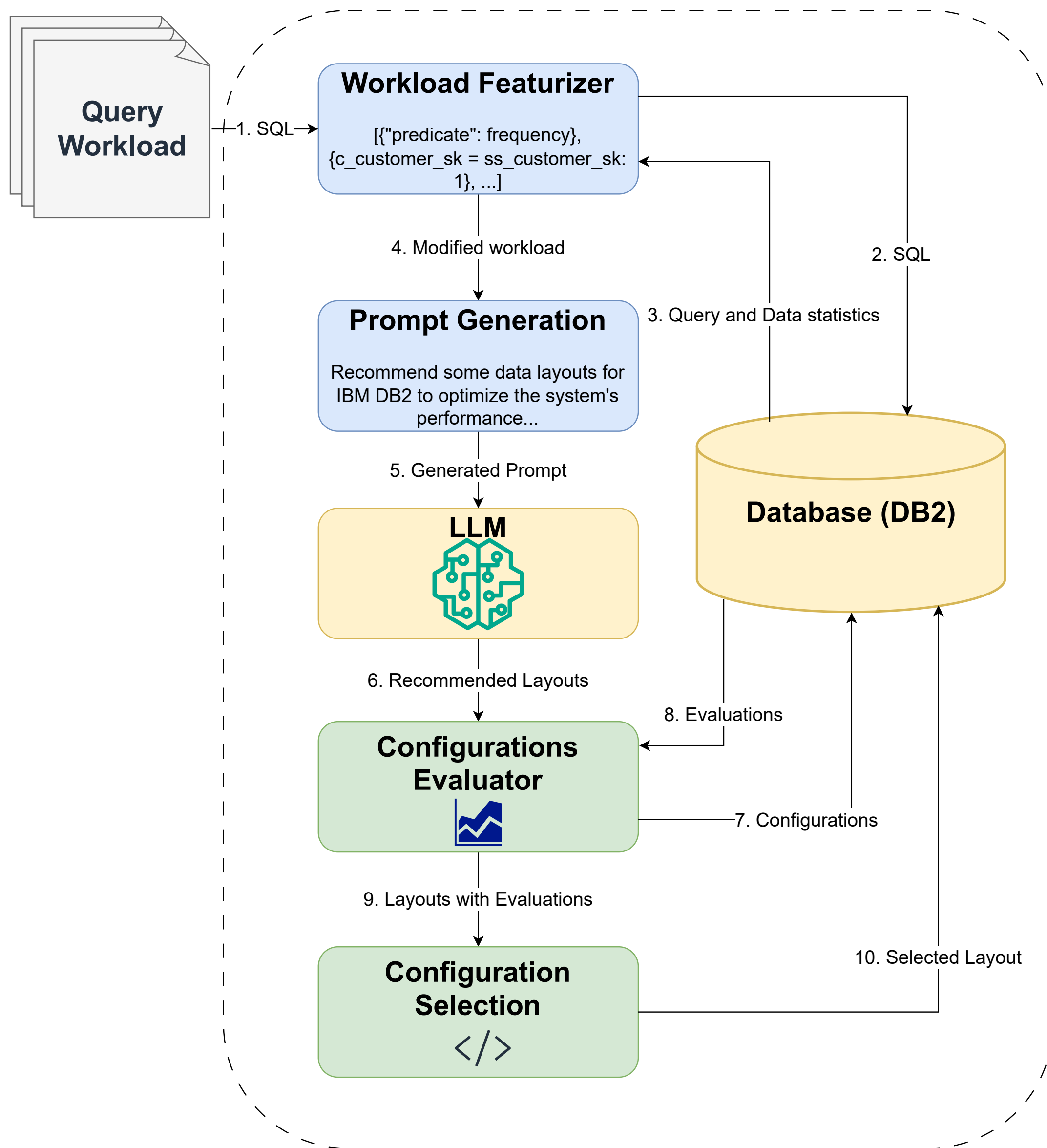


Figure 4.1: Architectural overview of LayZ.

seen in Figure 1.1).

To start the layout recommendation process from our system, a data layout recommendation request is made with a target workload (*Step* 1). The workload featurizer component processes the workload with respect to its current layout and configurations, generating a QEP for each query and statistics about the entire dataset (*Step* 2 and 3). Some information we extract from the QEP includes the predicates, join keys, and cardinality of base tables. These statistics include hardware information and table sizes. This information is then passed into the prompt generation component (*Step* 4).

After the component generates the necessary information into a succinct format, the prompt generator then featurizes all this information and generates a prompt out of the information for the LLM to execute (*Step* 5). The LLM will then execute the prompt with its given information. We prompt the LLM n times to get n configurations. Each configuration includes a full layout recommendation. We give it multiple rules that it must adhere to, such as the possible data layout options that the database supports. The LLM outputs multiple data layout configurations for the target workload. (*Step* 6). The multiple layout configurations are then evaluated. Since there might be duplicate configurations for similar queries, we eliminate those to avoid redundancy. Since running the queries exhaustively might take an exorbitant amount of time, we create a uniform sample of the database and evaluate our configurations on the sample database (*Step* 7 and 8). Each configuration will have an associated estimated query runtime based on our cost model. These evaluated configurations are then passed to a selection component (*Step* 9). After each configuration has a cost associated with it, we then select the most impactful configurations to apply to the

database. Currently, we determine the most impactful configurations by having the user choose an integer k where we select the top k configurations determined by our cost model that are generated from the LLM. After the configurations are selected, we generate the appropriate database code to apply the ordering (*Step* 10). The DBA then validates final data layout recommendations and applies the appropriate layouts.

4.2 Workload Featurizer

The typical workflow for using an LLM to recommend data layouts without `LayZ` would involve the user asking an LLM to determine the optimal configuration of the data layout for each table in the database for a given query workload. The user does not have any specific metric to judge optimality by, and gives it the entire query workload. The query workload is gigantic, and without an enormous amount of hardware resources, the LLM will not run. Let's say that the user has enough resources for the LLM, then there would be a huge amount of tokens to process, which would significantly increase the cost of using the LLM. To avoid this, the user selects a subset of the query workload to give to the LLM. This relies on the user's expertise to select representative queries of the query workload, which might not always be perfect and does not have any metadata, thus missing some crucial details that would impact data layout recommendations.

Wanting better recommendations that take into account more than just the queries, the user uses `LayZ` to get a data layout recommendation, giving it the same queries as it did to the LLM. `LayZ` leverages `IBM Db2`'s explain facility to get the QEPs from each query in the query workload, extracts all the predicates involved in the table

```
Recommend data layouts for IBM DB2 LUW to optimize the system's performance. Such data layouts include weighted z-ordering. Weighted z-ordering is z-order with a configuration on the number of bits to assign each value. Each table will have their own weighted z-order using its own columns.
Provide a response that contains the configuration for a weighted z-ordering. For example, movie_companies.company_type_id: 1 means take 1 bit from movie_companies each round of the bit interleaving in z-order.

The workload runs on a system with the following specs:
CPU Speed: 6.297924e-08
Buffer Pool size: 187958
Sort Heap size: 100000
Database Heap size: 5725

The following are all the predicates and their frequencies involved in the query workload in the format "predicate [count: ?]":
d_month_seq BETWEEN '?' AND '?' + '?' AND cs_ship_date_sk = d_date_sk AND cs_warehouse_sk = w_warehouse_sk AND cs_ship_mode_sk = sm_ship_mode_sk AND cs_call_center_sk = cc_call_center_sk AND cs_list_price BETWEEN '?' AND '?' AND sm_type = '?' AND cc_class = '?' AND w_gmt_offset = -'?' [count: 1]

Mapping of table and their columns:
catalog_sales: cs_list_price, cs_ship_date_sk, cs_warehouse_sk, cs_call_center_sk, cs_list_price
date_dim: d_month_seq, d_date_sk
warehouse: w_warehouse_sk, w_gmt_offset
shipping_mode: sm_ship_mode_sk, sm_type

call center: cc_call_center_sk
Table cardinality:
catalog_sales: 1440048277 rows
date_dim: 73049 rows
warehouse: 20 rows
shipping_mode: 20 rows
call center: 42 rows
```

Figure 4.2: Example Prompt

scans, the join columns, hardware information, and table sizes. `LayZ` then featurizes and encodes this information in a prompt for the LLM to process. This preprocessing reduces the number of tokens, thus reducing the resource usage and monetary cost of using the LLM. An example prompt is shown in Figure 4.2. We can see that the prompt only includes the predicates, hardware information, and table information.

All of this information is used to instruct the LLM on how to interpret and tune the query workload concerning data layouts. The user is now able to efficiently get a data layout recommendation for specific tables for the given query workload. These data layouts are able to interact with each other to get better results than an individual data layout on a table. Some of these data layouts have tunable parameters, such as weighted Z-ordering, and the LLM can adjust and give the weights for the weighted Z-ordering too. In short, `LayZ` can preprocess a query workload such that the input

size is reduced before giving it to the LLM while maintaining key features of the query workload and database metadata.

4.3 Prompt Generator

Once all the interesting characteristics are extracted from the QEP, the prompt generator embeds these characteristics into a prompt template. Additionally, hardware information and memory-related knobs, including buffer pool and sortheap size, are embedded into the prompt.

For prompts, the LLM should have a sense of the schema of our dataset, so we embed a table mapping where each column is mapped to its respective table. We attempted different variations of displaying the table schema, such as a table of all the table names and their columns; however, there was no noticeable impact on the performance. Furthermore, the LLM needs a sense of the respective sizes of each of the tables to determine which tables will significantly impact performance compared to the other tables; thus, we embed the table cardinalities of the involved tables.

4.4 Configuration Evaluation

To finally recommend a given configuration, we need to evaluate the candidate configurations and choose the best one according to the goals of the user.

At the beginning of the evaluation process, we create a uniform sample dataset from the full dataset. Evaluating the full query workload would be too time-intensive due to the number of candidates that need to be evaluated and the scale of some workloads. To evaluate the candidates, we have a cost model based on the Gao's

`Bayesian Optimization` system [8] to determine the effectiveness of a candidate. In their approach, their cost model is dependent on number of rows and columns scanned multiplied by the time it takes to read a single row of a single column. They do not consider the time it takes to read each data type. Therefore we modify our cost model to accommodate this:

$$\text{cost} = r \times \sum_{j=1}^{c} w_{\tau(j)}$$

where r is the number of rows scanned, c is the number of columns used by the query, $\tau(j)$ denotes the data type of column j, and $w_{\tau(j)}$ is the scan cost of the column based on the data type. Since there can be a large amount of data in the database, especially in a cloud setting, we utilize the `IBM Db2` synopsis tables on the sampled database to determine the number of rows scanned. Each row in the synopsis table has a metadata column that contains the number of rows in that particular data block, as shown in Table 2.1. We determine the number of rows scanned by seeing which data blocks satisfy the predicate and how many rows are in that data block. This is done for each candidate configuration to generate the cost. Each configuration will have an associated cost that determines its effectiveness.

4.5 Configuration Selection

The candidate configurations must now be selected for a final recommendation from the advisor. After each configuration has a cost associated with it, we then select the most impactful configurations to apply to the database. Currently, we determine the most impactful configurations by having the user choose an integer k, where we select

the top k configurations generated from the LLM. This number can be automatically determined through other techniques, such as a learned model that approximates the best k for a target workload; however, our current system does not do this. Some of the recommendations that are not applied to the database can be used for other physical design structures, such as MVs. For example, if $k = 3$, then the top choice would be the primary layout for the database, and the second top choice can be applied to a MV.

Chapter 5

System Demonstration

To showcase the `LayZ` system, we developed an interactive web application that lets users use a data layout advisor to be recommended data layouts to boost performance and see how `LayZ`'s recommendations aid users in creating other physical design structures to enhance the complex analytical workload performance.

5.1 Query-Aware Data Layout Recommendations

Queries that use table scans and filters as a large part of their performance benefit greatly from data layouts. For example, if the user executes the query in Listing 5.1, then there would be intensive table scans and filtering involved in the query.

```sql
SELECT min(i_brand_id), min(i_manufact_id),
    min(ss_ext_sales_price)
FROM date_dim, store_sales, item, customer,
    customer_address, store
WHERE d_date_sk = ss_sold_date_sk
   and ss_item_sk = i_item_sk
   and ss_customer_sk = c_customer_sk
   and c_current_addr_sk = ca_address_sk
   and ss_store_sk = s_store_sk
   and i_category  = 'Home'
   and d_year=2001
   and d_moy = 10
   and substring(ca_zip,1,5) <> substring(s_zip,1,5)
   and ca_state  = 'OR'
   and c_birth_month = 12
   and ss_wholesale_cost BETWEEN 51 AND 71;
```

Listing 5.1: Modified query based on DSB Query 19_SPJ

Even just modifying the data layout of the largest table in the query (i.e store_sales) would yield benefits as large as 90% as shown in Table 5.1. In Table 5.1, we do a per-query performance breakdown for selected DSB queries. We normalize by the random layout, because that is a common layout in databases, when no layout is selected.

We also compare the performance between Gao's `Bayesian Optimization` (BO) [8] and the LLM method. Gao's `Bayesian Optimization` utilizes Bayesian optimiza-

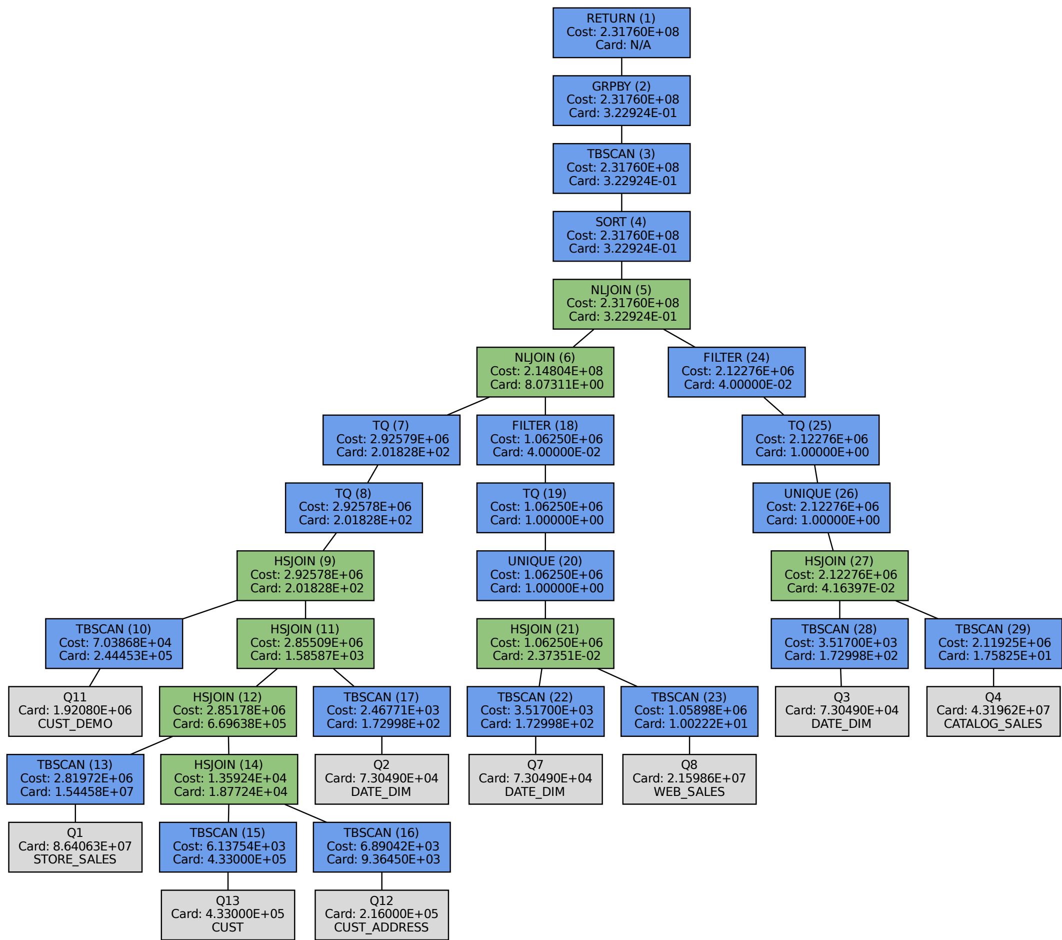


Figure 5.1: Query Execution Plan for DSB Q_{69}.

Query	Runtime (relative to Random layout)		
	Random (Baseline)	LayZ	Bayesian Optimization
q10	1.0	0.20	0.21
q19_spj	1.0	0.31	0.30
q19	1.0	0.45	0.44
q92	1.0	0.05	0.07
q58	1.0	0.04	0.06
q69	1.0	0.06	0.09
q25_spj	1.0	0.06	0.07
q25	1.0	0.07	0.08
q50	1.0	0.10	0.11
q99	1.0	0.10	0.16

Table 5.1: Per-query template breakdown of runtime in DSB Workload

tion to search over Z-order configurations for a target table. They have a custom cost model described in Chapter 4.4, which is their objective function that is passed to the Bayesian optimization algorithm. Our LLM method can recommend configurations for multiple tables at a time, and does not have the same training time as the BO method. We can see that in Figure 5.2 that there are multiple tables recommended by the LLM. We analyze the information collected from the QEP in Figure 5.1, such as the predicates in the table scans, table cardinalities, and cost estimates. The QEP enhances the LLM's output because it gives information on the dataset, as opposed to the LLM attempting to infer it using external information.

LayZ: LLM-directed Data Layout Advisor

Query Workload

```
select
cd_gender, cd_marital_status, cd_education_status, count(*) cnt1,
cd_purchase_estimate, count(*) cnt2, cd_credit_rating, count(*) cnt3
from
customer c,customer_address ca,customer_demographics
where
c.c_current_addr_sk = ca.ca_address_sk and ca_state in ('KS','LA','OK') and
cd_demo_sk = c.c_current_cdemo_sk and cd_marital_status in ('W', 'S', 'S')
and cd_education_status in ('Unknown', '4 yr Degree') and
  exists (select *
          from store_sales,date_dim
          where c.c_customer_sk = ss_customer_sk and
```

Upload SQL File

CHOOSE FILE

☑ Load content into textbox?

\# of Configurations: 1

GENERATE

Weighted Z-ORDER Configurations

Table: store_sales

Column	Weight
ss_list_price	2

Table: web_sales

Column	Weight
ws_list_price	2

Table: catalog_sales

Column	Weight
cs_list_price	2

Table: customer_address

Column	Weight
ca_state	5

Table: customer_demographics

Column	Weight
cd_marital_status	4
cd_education_status	4

Table: date_dim

Column	Weight
d_year	3
d_moy	3

Figure 5.2: Main Screen of the `LayZ` Layout Advisor.

5.2 Joint Interaction with Materialized View (MV)s and Data Layouts

Not all queries benefit from a single ordering. For example, if queries do not use the Z-ordered columns, there will be unnecessary ordering overhead, or if the weighted Z-ordering has suboptimal weights, such as assigning high weights to low selectivity columns and low weights to high selectivity columns, then queries might increase in runtime. To combat this issue, users can create materialized views to increase performance for the regressed queries.

The user runs `LayZ` and gets a data layout for their tables in the database. The user runs their query workload and notices some regressions in certain queries. The user is then able to create a materialized view for their query and give the rewritten query to `LayZ` to recommend a data layout for the materialized view.

`LayZ` assists in the recommendation of an MV because it allows a more diverse range of QEPs to be used when processing a query. For instance, the query in Figure 5.2 benefits from a specific ordering on the store_sales table, however that same ordering can cause a regression in the query in listing 5.1. An MV recommendation could be a subset of the store_sales table in a different layout. This allows for many more queries to be processed faster, using different data layouts.

In Listing 5.4, we have an example of a materialized view that has the potential to benefit many queries, especially in the TPC-DS workload [58]. For example, the MV can match TPC-DS Q_4 and Q_{10}, specifically matching the snippets in Listings 5.2 and 5.3. Currently, there is no layout applied to the MV, leaving lots of room for improvement. To resolve this, we can z-order the MV by the join key columns

to increase the performance of the query. The TPC-DS workload contains many predicates involving the *ss_customer_sk*, and *ss_sold_date_sk* columns. If a weighted z-ordering on those two columns is applied to the MV, then synopsis tables can reduce the number of rows accessed even further, improving the query processing performance immensely.

```
...
SELECT ...,
       SUM(((ss_ext_list_price-
       ss_ext_wholesale_cost-
       ss_ext_discount_amt)+
       ss_ext_sales_price)/2) year_total, ...
FROM customer
     ,store_sales
     ,date_dim
WHERE c_customer_sk = ss_customer_sk
   AND ss_sold_date_sk = d_date_sk
...
```

Listing 5.2: Snippet of TPC-DS Q_4

```
...
SELECT *
FROM store_sales, date_dim
WHERE c.c_customer_sk = ss_customer_sk AND
    ss_sold_date_sk = d_date_sk AND
    d_year = 2000 AND
    d_moy BETWEEN 1 AND 1+3
...
```

Listing 5.3: Snippet of TPC-DS Q_{10}

```sql
SELECT
    ss_customer_sk,
    ss_sold_date_sk,
    sum(ss_net_paid) sum_1,
    sum(ss_quantity * ss_sales_price) sum_2,
    sum(ss_ext_sales_price) sum_3,
    sum(
        ((ss_ext_list_price - ss_ext_wholesale_cost
        - ss_ext_discount_amt) + ss_ext_sales_price) / 2
    ) sum_4,
    sum(ss_ext_list_price - ss_ext_discount_amt) sum_5,
    count(*) number_sales
FROM
    store_sales
GROUP BY
    ss_customer_sk,
    ss_sold_date_sk
```

Listing 5.4: Example Query for a Materialized View

Chapter 6

Experimental Evaluation

We perform an experimental validation of LayZ, showcasing its efficiency and effectiveness. The experiments were run on a virtual machine with an Intel(R) Xeon(R) Gold 6442Y CPU using 16 cores and 32GB of RAM, running IBM Db2 Enterprise Edition. We evaluate LayZ on the DSB [14] benchmark. IBM Db2 is column-organized and DSB is 1000GB of data. All experimental results for LayZare with respect to the use of OpenAI o3 unless otherwise stated. In this chapter, we present the results of experiments that compare LayZ performance on per-query and query workload execution times, LayZ with the Gao's Bayesian Optimization system [8] and random order, LayZ with different LLMs, and LayZ with different knob settings. Knob settings are configurable parameters that control some aspect of database behavior or resource allocation.

6.1 Query and Workload Level Tuning

We evaluate the LayZ system against the Gao's Bayesian Optimization system and random order over the DSB workload. To demonstrate LayZ's tuning effectiveness, we evaluate LayZ against Gao's Bayesian Optimization and random order on the query template level and query workload level. Each query template will have 20 distinct queries.

Exp-1: Query Workload Level. We first evaluate LayZ's efficiency in tuning on a query workload level. This is achieved by feeding the entire query workload to LayZ. In our experiments, the weighted Z-order configurations recommended by systems tend to place higher weight on join keys compared to non-join keys. As illustrated in Figure 6.1, LayZ's recommended layout configurations result in high performance gains over the DSB workload. On DSB, LayZ reduces total execution time by 8.6% compared to Gao's Bayesian Optimization and by 69.6% compared to random order.

The execution times were close in magnitude for Gao's Bayesian Optimization and LayZ however, we note that the time to get the layout recommendations was much longer using Gao's Bayesian Optimization due to BO needing to train for 150 iterations.

LayZ's recommendations when given an entire query workload tended to put more weight on the join keys of the query workload. This aligns with the fact that joins are some of the most expensive operations in a database so the LLM is attempting to reduce the cost of joins by proposing certain layouts.

Exp-2: Query Level. We next evaluate how the LayZ system tunes on a query level basis. This is achieved by feeding specific queries to LayZ. We select specific queries

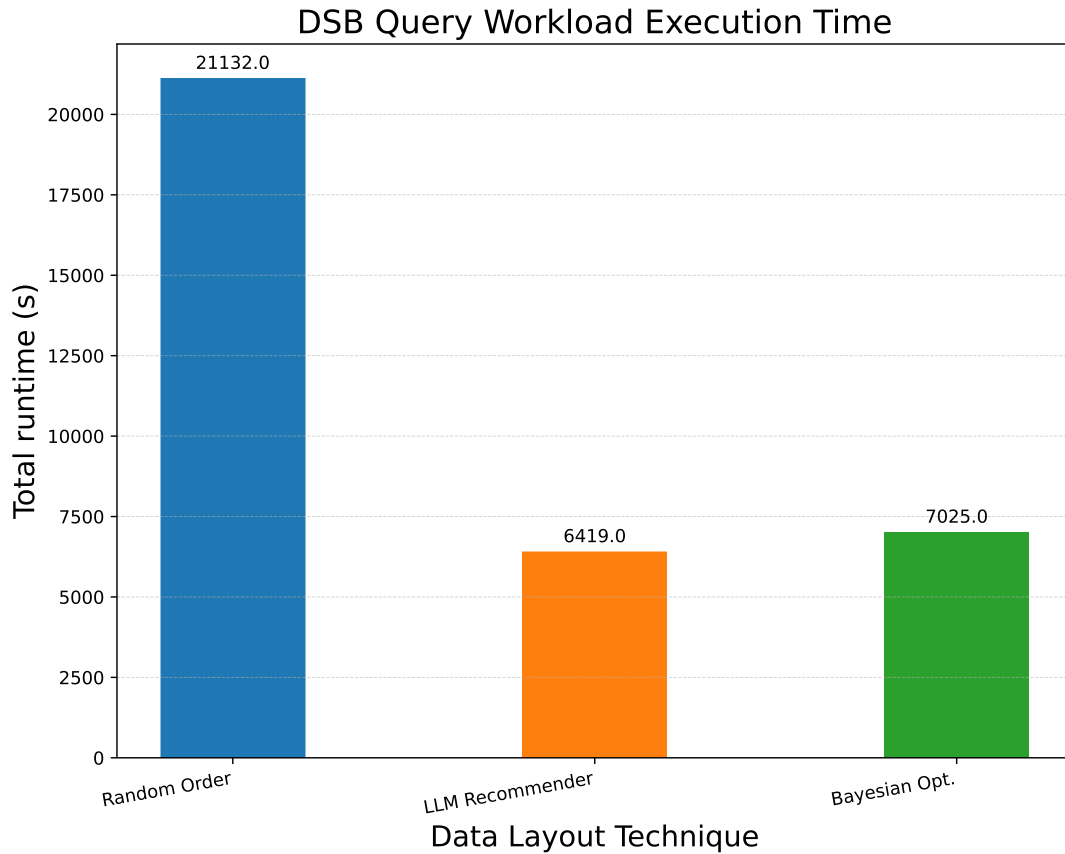


Figure 6.1: Workload Level Tuning.

from DSB that involve a large number of rows scanned in the table scan operations on the fact tables of the DSB schema. As shown in Figure 6.2, `LayZ` outperforms Gao's `Bayesian Optimization` and random order over the DSB workload. On DSB, `LayZ` has a quicker execution time in all instances of the random order and quicker runtimes than Gao's `Bayesian Optimization` on 8 out of 10 selected queries.

We notice that per-query recommendations tend to include weights on the measure columns in fact tables (e.g SS_LIST_PRICE). Now that there are less columns to

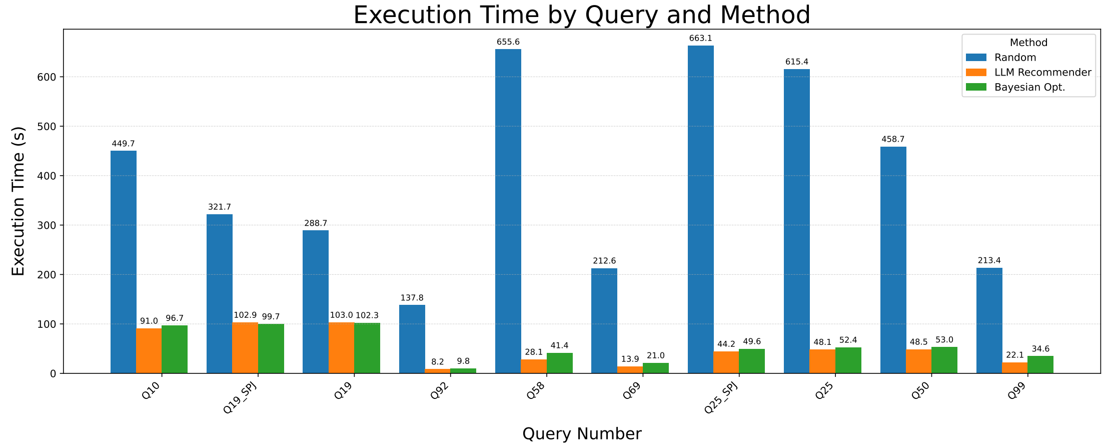


Figure 6.2: Query level tuning.

consider compared to the query workload, the LLM can tailor their recommendations more to the specific query. This is beneficial as too many columns included in the Z-order will not achieve effective data skipping and too little columns included in the Z-order will perform the same as a single-key column sort.

6.2 Large Language Model Comparison

Exp-3: LLM System Recommendation Comparison We next evaluate the effectiveness of different LLMs for data layout tuning by feeding the LLM the same input prompt. We utilize the models Open AI o3 model, Granite 4.0 tiny preview model [59], and the GPT-4o model to make our comparisons.

As we can see in Figure 6.3, o3 model outperforms the other models in recommending layout tuning configurations. This can be because o3 is OpenAI's is their most powerful reasoning model.

Due to the memory constraints, we were unable to provide the query workload

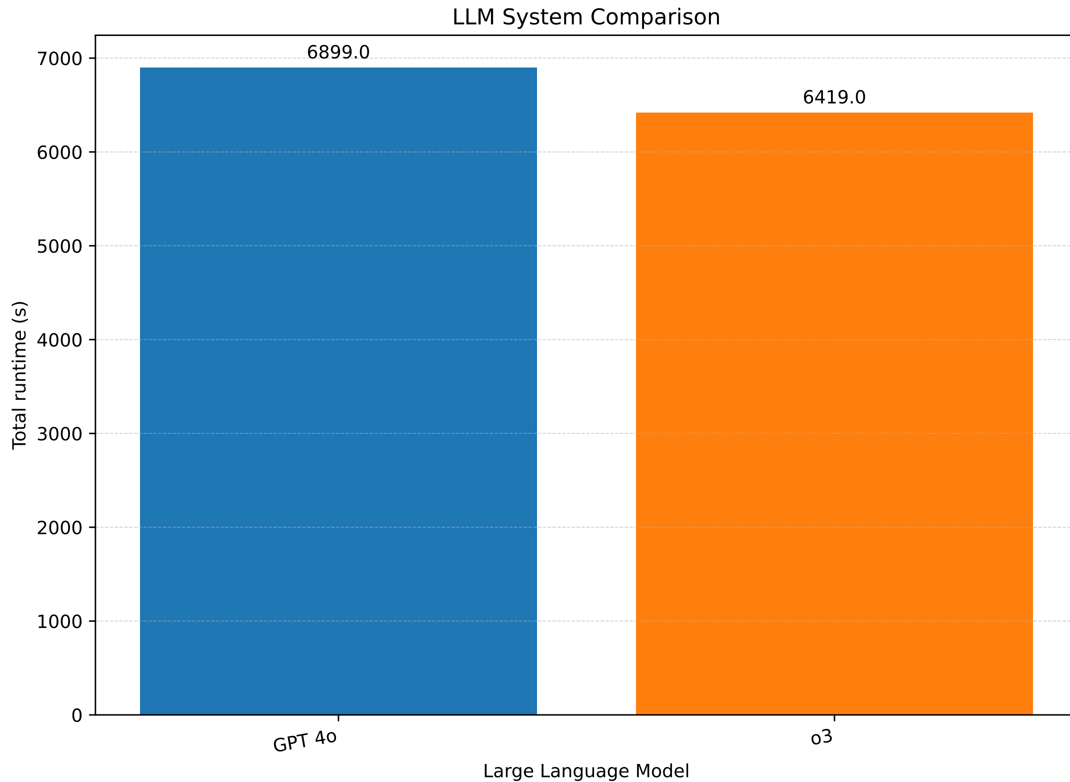


Figure 6.3: LLM System Comparison.

execution times for the Granite 4.0 tiny preview model, however, we present Figure 6.4 as an example an individual query template runtimes, specifically DSB query 99. This shows how preprocessing is important for LLMs with limited resources.

6.3 Database Knob Evaluation

Exp-4: Impact of Knobs To study the inclusion of knobs into the data layout tuning, we evaluate two approaches over the DSB workload: with *default knobs* and *`IBM` documentation recommended knobs*. The default knobs are set by `IBM Db2` when

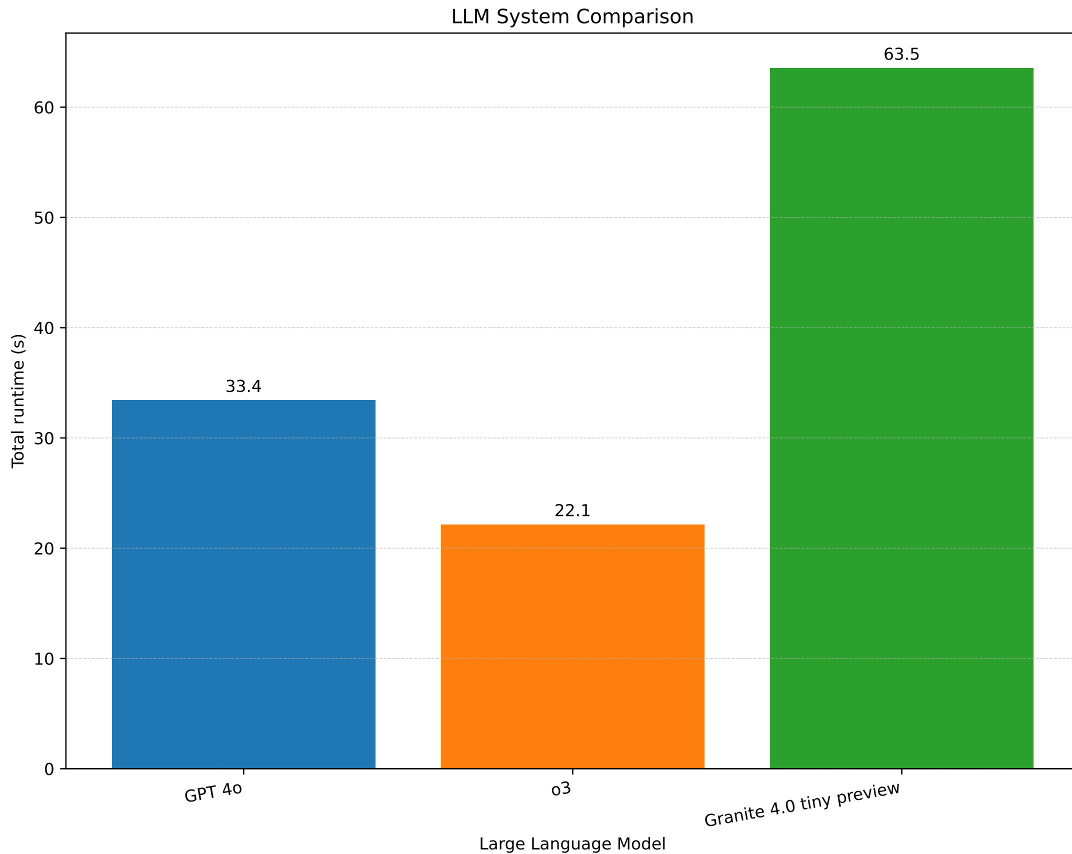


Figure 6.4: LLM System Comparison on DSB Q99.

you create a column-organized database. The documentation recommended knobs [60] are the configurations recommended by `IBM` when setting up a column-organized database.

We mainly tune the sortheap heap. We choose the sortheap knob because this knob controls the maximum number of memory pages that an operation in the QEP is allowed to use. As a result, changes in this parameter can affect the speed and number of rows that are processed in an operation. Therefore, this can impact the

effectiveness of data layouts.

We measure the changes through query execution time. As we can see in Figure 6.5, sortheap has a noticeable impact on query execution time.

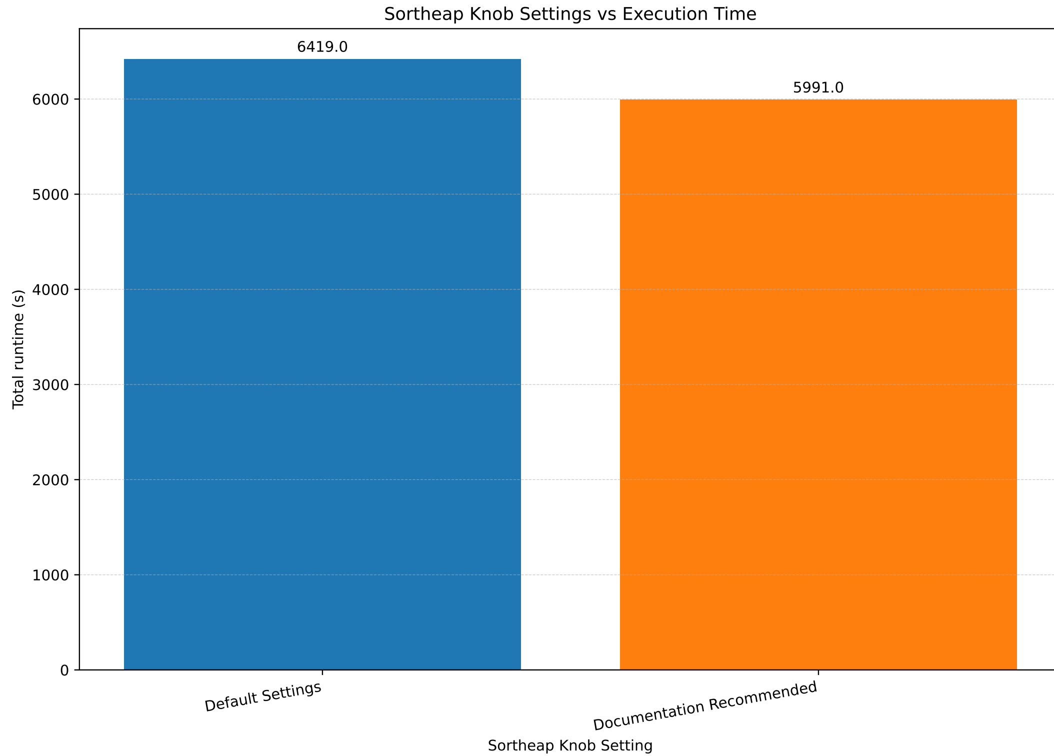


Figure 6.5: Sortheap Knob Setting vs Execution Time.

6.4 Layout Recommendation Times

Exp-5: `LayZ` versus Gao's `Bayesian Optimization` Recommendation time Finally, we compare the time to get the finalized data layout recommendation from the `LayZ` and Gao's `Bayesian Optimization` systems. We compare the recommendation

times on a query workload level. We configure BO to run for 150 iterations. Note that for Gao's `Bayesian Optimization` we can only get the weights for one table at a time; meanwhile, `LayZ` can recommend a holistic configuration for all tables in the dataset. As a result, we use the total Gao's `Bayesian Optimization` recommendation time for the 3 largest tables by cardinality in the DSB dataset.

As we can see in Figure 6.6, `LayZ` 's recommendation time outperforms Gao's `Bayesian Optimization` in recommending layout tuning configurations by over 150%. This is because the Bayesian Optimization method requires multiple iterations to explore and exploit the massive configuration space. LLMs already have lots of pre-trained knowledge within them and can utilize this knowledge to recommend "reasonable" layout configurations with quick processing times.

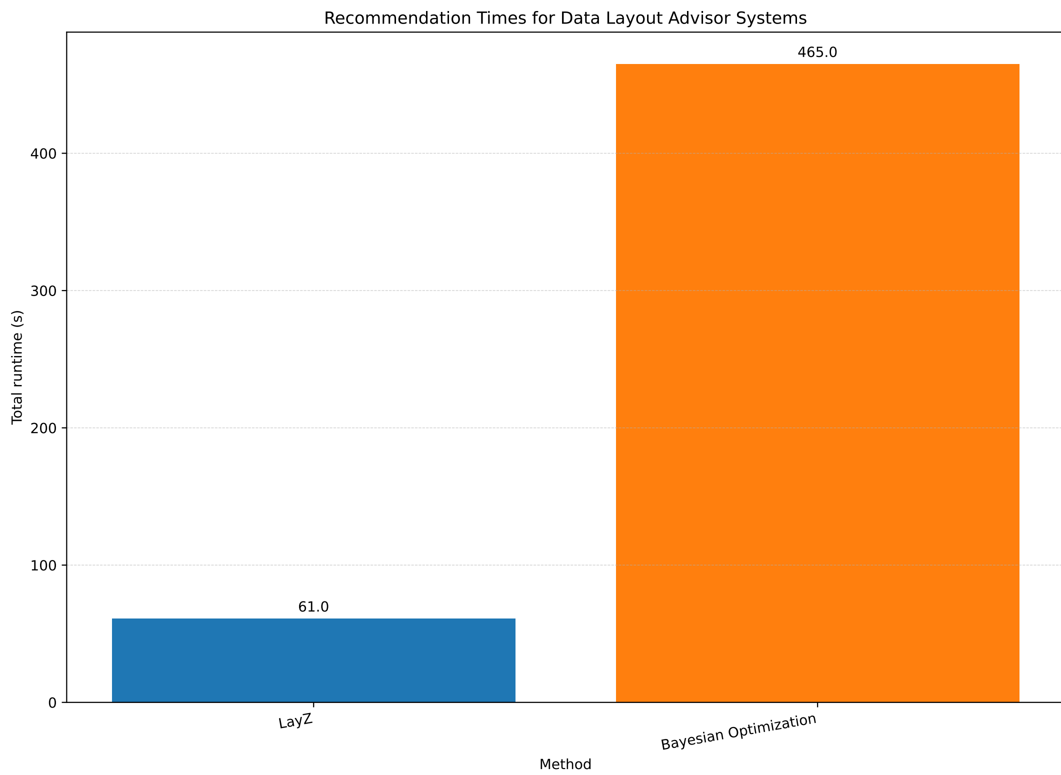


Figure 6.6: Data Layout Recommendation times of `LayZ` vs other.

Chapter 7

Conclusion

7.1 Conclusion

In this work, we present LayZ, a large language model (LLM)- directed data layout advisor designed to address the growing complexity of tuning physical layouts in analytical databases, such as IBM Db2. As modern workloads span numerous tables with complex joins, filters, and skewed distributions in a cloud environment, traditional heuristic-based layout strategies fall short. LayZ addresses this challenge by integrating LLM-based reasoning with the database's query execution infrastructure at a reduced cost compared to existing layout advisors.

Our system preprocesses query workloads using query execution plans to extract meaningful features such as predicates, join keys, data distributions, and table statistics, and encodes them into compact natural language prompts for an LLM. These prompts allow the model to make intelligent layout recommendations, including configurations for weighted Z-ordering, while dramatically reducing the computational and monetary overhead typically associated with large language models.

LayZ offers a cost-model–driven evaluation layer that selects the most promising candidate layouts for a target workload. It further distinguishes itself by supporting layout recommendations not only for base tables but also for materialized views, enabling recovery from performance regressions and extending layout optimization across multiple layers of the data processing stack.

Through a series of real-world scenarios based on the DSB benchmark, we demonstrate how LayZ adapts to diverse query patterns and data characteristics, achieving performance gains that rival and in some cases surpass industry expert layout strategies. Moreover, LayZ's ability to recommend multi-table layouts with minimal human effort significantly broadens the scope and practicality of data layout tuning.

Ultimately, LayZ represents a new class of intelligent tuning tools that leverage the power of LLMs not just for query rewriting or knob tuning, but for deep physical design optimization. It opens promising avenues for automated database tuning where human expertise is limited, workloads evolve, and performance matters.

7.2 Future Work

LayZ is designed as one piece of a greater self-tuning database system. LayZ currently focuses on optimizing data layouts, including weighted Z-ordering, at the table and materialized view level. A natural extension is to broaden the system's scope to include other physical design elements such as index selection, materialized view design, and buffer pool configuration. A unified LLM-based advisor could reason across these layers to produce holistic physical design recommendations.

Currently, LayZ operates reactively, recommending layouts based on the current query workload. However, in dynamic environments such as enterprise data ware-

houses or cloud-native analytics platforms, workloads often shift based on time-of-day, user behavior, or business cycles. By integrating query workload forecasting to predict future query patterns, `LayZ` could anticipate shifts in workload characteristics and recommend layouts ahead of time. For example, if forecasted queries involve different tables, filters, or access patterns than the current workload, `LayZ` could precompute candidate layouts or materialized views before performance degradation occurs.

The LLMs in `LayZ` operate based on static prompt engineering and pre-trained knowledge. An adaptive variant of `LayZ` could incorporate feedback from observed query performance to iteratively refine its recommendations. This could involve fine-tuning on layout-performance pairs or reinforcement learning over diverse query workloads to improve over time. Another promising direction is the integration of retrieval-augmented generation (RAG) to enhance the quality, accuracy, and generalizability of `LayZ`'s layout recommendations. Since `LayZ` currently relies on pre-trained LLMs and structured prompt templates, these models are limited to static knowledge and lack access to system-specific documentation, prior tuning decisions, or institutional best practices. RAG allows `LayZ` to dynamically retrieve relevant context such as past tuning logs, workload summaries, database documentation, and historical query plans from similar queries. This retrieved content can be embedded into the prompt, enabling the LLM to generate recommendations grounded in specific, relevant, and up-to-date system behavior.

The cost model used to rank layout configurations is currently rule-based. Replacing or augmenting it with learned models trained on real workload runtimes could improve accuracy and generalization. In addition, improving explainability, that is,

why a specific layout was chosen, would build trust in the system and support debugging and auditing.

References

[1] D. Comer, “Ubiquitous b-tree,” *ACM Comput. Surv.*, vol. 11, no. 2, pp. 121–137, Jun. 1979, ISSN: 0360-0300. DOI: 10.1145/356770.356776. [Online]. Available: https://doi.org/10.1145/356770.356776.

[2] P. E. O’Neil, “Model 204 architecture and performance,” in *Proceedings of the 2nd International Workshop on High Performance Transaction Systems*, Berlin, Heidelberg: Springer-Verlag, 1987, pp. 40–59, ISBN: 3540510850.

[3] P.-Å. Larson and H. Z. Yang, “Computing queries from derived relations,” in *Proceedings of the 11th International Conference on Very Large Data Bases - Volume 11*, ser. VLDB ’85, Stockholm, Sweden: VLDB Endowment, 1985, pp. 259–269.

[4] G. Moerkotte, “Small materialized aggregates: A light weight index structure for data warehousing,” in *Proceedings of the 24rd International Conference on Very Large Data Bases*, ser. VLDB ’98, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998, pp. 476–487, ISBN: 1558605665.

[5] S. Agrawal, S. Chaudhuri, L. Kollar, A. Marathe, V. Narasayya, and M. Syamala, “- database tuning advisor for microsoft sql server 2005,” in *Proceedings 2004 VLDB Conference*, M. A. Nascimento, M. T. Özsu, D. Kossmann,

R. J. Miller, J. A. Blakeley, and B. Schiefer, Eds., St Louis: Morgan Kaufmann, 2004, pp. 1110–1121, ISBN: 978-0-12-088469-8. DOI: https://doi.org/10.1016/B978-012088469-8.50097-8. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780120884698500978.

[6] D. C. Zilio, J. Rao, S. Lightstone, *et al.*, "Db2 design advisor: Integrated automatic physical database design," in *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, ser. VLDB '04, Toronto, Canada: VLDB Endowment, 2004, pp. 1087–1097, ISBN: 0120884690.

[7] A. Documentation, *Amazon redshift advisor recommendations*, https://docs.aws.amazon.com/redshift/latest/dg/advisor-recommendations.html, Accessed: 2025-07-01, 2025.

[8] J. Gao, J. Ding, S. Sudhir, and S. Madden, "Learning bit allocations for z-order layouts in analytic data systems," in *Proceedings of the Seventh International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, ser. aiDM '24, Association for Computing Machinery, 2024, ISBN: 9798400706806. DOI: 10.1145/3663742.3663975. [Online]. Available: https://doi.org/10.1145/3663742.3663975.

[9] V. Giannakouris and I. Trummer, "λ-tune: Harnessing large language models for automated database system tuning," *Proceedings of the ACM on Management of Data*, vol. 3, no. 1, pp. 1–26, 2025.

[10] I. Trummer, "DB-BERT: A database tuning tool that "reads the manual"," in *Proceedings of the 2022 International Conference on Management of Data*,

ser. SIGMOD '22, ACM, 2022, pp. 190–203. DOI: 10.1145/3514221.3517843. [Online]. Available: https://doi.org/10.1145/3514221.3517843.

[11] R. M. Perera, B. Oetomo, B. I. P. Rubinstein, and R. Borovica-Gajic, "Hmab: Self-driving hierarchy of bandits for integrated physical database design tuning," *Proc. VLDB Endow.*, vol. 16, no. 2, pp. 216–229, Oct. 2022, ISSN: 2150-8097. DOI: 10.14778/3565816.3565824. [Online]. Available: https://doi.org/10.14778/3565816.3565824.

[12] D. Van Aken, A. Pavlo, G. J. Gordon, and B. Zhang, "Automatic database management system tuning through large-scale machine learning," in *SIGMOD*, 2017, pp. 1009–1024.

[13] I. D. Documentation, *Db2 synopsis tables*, https://www.ibm.com/docs/en/db2/12.1?topic=organization-synopsis-tables, Accessed: 2025-07-01, 2025.

[14] B. Ding, S. Chaudhuri, J. Gehrke, and V. Narasayya, "Dsb: A decision support benchmark for workload-driven and traditional database systems," *Proc. VLDB Endow.*, vol. 14, no. 13, pp. 3376–3388, Sep. 2021, ISSN: 2150-8097. DOI: 10.14778/3484224.3484234. [Online]. Available: https://doi.org/10.14778/3484224.3484234.

[15] Anthropic, *Anthropic documentation*, https://www.anthropic.com/pricing, Accessed: 2021-10-04, 2022.

[16] OpenAI, *Open ai documentation*, https://openai.com/api/pricing/, Accessed: 2021-10-04, 2022.

[17] T. Kim, Y. Wang, V. Chaturvedi, *et al.*, *Llmem: Estimating gpu memory usage for fine-tuning pre-trained llms*, 2024. arXiv: 2404.10933 [cs.AI]. [Online]. Available: https://arxiv.org/abs/2404.10933.

[18] I. D. Documentation, *Db2 query optimization process*, https://www.ibm.com/docs/en/db2/12.1.0?topic=optimization-sql-xquery-compiler-process, Accessed: 2025-07-01, 2025.

[19] I. D. Documentation, *Db2 registry and environment variables*, https://www.ibm.com/docs/en/db2/11.5?topic=variables-registry-environment, Accessed: 2021-10-04, 2022.

[20] I. D. Documentation, *Db2 optimization classes*, https://www.ibm.com/docs/en/db2/12.1.0?topic=plans-optimization-classes, Accessed: 2025-07-01, 2025.

[21] S. K. Rahimi and F. S. Haug, *Distributed database management systems: A Practical Approach*. John Wiley & Sons, 2015.

[22] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, "Access path selection in a relational database management system," in *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '79, Boston, Massachusetts: Association for Computing Machinery, 1979, pp. 23–34, ISBN: 089791001X. DOI: 10.1145/582095.582099. [Online]. Available: https://doi.org/10.1145/582095.582099.

[23] A. Bianchi, A. Chai, V. Corvinelli, P. Godfrey, J. Szlichta, and C. Zuzarte, "Db2une: Tuning under pressure via deep learning," *Proc. VLDB Endow.*,

vol. 17, no. 12, pp. 3855–3868, 2024. DOI: 10.14778/3685800.3685811. [Online]. Available: https://doi.org/10.14778/3685800.3685811.

[24] C. Henderson, S. Bryson, V. Corvinelli, *et al.*, "Blutune: Query-informed multi-stage ibm db2 tuning via ml," in *Proceedings of the 31st ACM International Conference on Information and Knowledge Management*, ser. CIKM '22, 2022, pp. 3162–3171, ISBN: 9781450392365. DOI: 10.1145/3511808.3557117. [Online]. Available: https://doi.org/10.1145/3511808.3557117.

[25] J. Zhang, Y. Liu, K. Zhou, *et al.*, "An end-to-end automatic cloud database tuning system using deep reinforcement learning," in *Proceedings of the 2019 International Conference on Management of Data*, ser. SIGMOD '19, ACM, 2019, pp. 415–432. DOI: 10.1145/3299869.3300085. [Online]. Available: https://doi.org/10.1145/3299869.3300085.

[26] S. Das, M. Grbic, I. Ilic, *et al.*, "Automatically indexing millions of databases in microsoft azure SQL database," in *Proceedings of the 2019 International Conference on Management of Data*, ser. SIGMOD '19, ACM, 2019, pp. 666–679. DOI: 10.1145/3299869.3314035. [Online]. Available: https://doi.org/10.1145/3299869.3314035.

[27] P. Negi, Z. Wu, A. Kipf, *et al.*, "Robust query driven cardinality estimation under changing workloads," *Proc. VLDB Endow.*, vol. 16, no. 6, pp. 1520–1533, 2023. DOI: 10.14778/3583140.3583164. [Online]. Available: https://www.vldb.org/pvldb/vol16/p1520-negi.pdf.

[28] J. Sun and G. Li, "An end-to-end learning-based cost estimator," *Proc. VLDB Endow.*, vol. 13, no. 3, pp. 307–319, 2019. DOI: 10.14778/3368289.3368296. [Online]. Available: http://www.vldb.org/pvldb/vol13/p307-sun.pdf.

[29] I. D. Documentation, *Db2 zone map*, https://www.ibm.com/docs/en/netezza?topic=ds-zone-maps, Accessed: 2025-07-01, 2025.

[30] S. Padmanabhan, B. Bhattacharjee, T. Malkemus, L. Cranston, and M. Huras, "Multi-dimensional clustering: A new data layout scheme in db2," in *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '03, San Diego, California: Association for Computing Machinery, 2003, pp. 637–641, ISBN: 158113634X. DOI: 10.1145/872757.872835. [Online]. Available: https://doi.org/10.1145/872757.872835.

[31] A. I. Documentation, *Apache iceberg internals*, https://www.dremio.com/resources/guides/apache-iceberg-an-architectural-look-under-the-covers/, Accessed: 2025-07-01, 2025.

[32] C. Liu, A. Pavlenko, M. Interlandi, and B. Haynes, "A deep dive into common open formats for analytical dbmss," *Proc. VLDB Endow.*, vol. 16, no. 11, pp. 3044–3056, Jul. 2023, ISSN: 2150-8097. DOI: 10.14778/3611479.3611507. [Online]. Available: https://doi.org/10.14778/3611479.3611507.

[33] X. Zeng, Y. Hui, J. Shen, A. Pavlo, W. McKinney, and H. Zhang, "An empirical evaluation of columnar storage formats," *Proc. VLDB Endow.*, vol. 17, no. 2, pp. 148–161, Oct. 2023, ISSN: 2150-8097. DOI: 10.14778/3626292.3626298. [Online]. Available: https://doi.org/10.14778/3626292.3626298.

[34] G. Li, X. Zhou, S. Li, and B. Gao, "Qtune: A query-aware database tuning system with deep reinforcement learning," *Proc. VLDB Endow.*, vol. 12, no. 12, pp. 2118–2130, 2019. DOI: 10.14778/3352063.3352129. [Online]. Available: http://www.vldb.org/pvldb/vol12/p2118-li.pdf.

[35] J. Ge, Y. Chai, and Y. Chai, "Watuning: A workload-aware tuning system with attention-based deep reinforcement learning," *J. Comput. Sci. Technol.*, vol. 36, no. 4, pp. 741–761, 2021. DOI: 10.1007/S11390-021-1350-8. [Online]. Available: https://doi.org/10.1007/s11390-021-1350-8.

[36] J. Wang, I. Trummer, and D. Basu, "UDO: universal database optimization using reinforcement learning," *Proc. VLDB Endow.*, vol. 14, no. 13, pp. 3402–3414, 2021. DOI: 10.14778/3484224.3484236. [Online]. Available: http://www.vldb.org/pvldb/vol14/p3402-wang.pdf.

[37] X. Zhang, H. Wu, Z. Chang, *et al.*, "Restune: Resource oriented tuning boosted by meta-learning for cloud databases," in *Proceedings of the 2021 International Conference on Management of Data*, ser. SIGMOD '21, ACM, 2021, pp. 2102–2114. DOI: 10.1145/3448016.3457291. [Online]. Available: https://doi.org/10.1145/3448016.3457291.

[38] B. Cai, Y. Liu, C. Zhang, *et al.*, "HUNTER: an online cloud database hybrid tuning system for personalized requirements," in *Proceedings of the 2022 International Conference on Management of Data*, ser. SIGMOD '22, ACM, 2022, pp. 646–659. DOI: 10.1145/3514221.3517882. [Online]. Available: https://doi.org/10.1145/3514221.3517882.

[39] D. V. Aken, D. Yang, S. Brillard, *et al.*, "An inquiry into machine learning-based automatic configuration tuning services on real-world database management systems," *Proc. VLDB Endow.*, vol. 14, no. 7, pp. 1241–1253, 2021. DOI: `10.14778/3450980.3450992`. [Online]. Available: `http://www.vldb.org/pvldb/vol14/p1241-aken.pdf`.

[40] D. Eriksson and M. Jankowiak, "High-dimensional bayesian optimization with sparse axis-aligned subspaces," in *Proceedings of the Thirty-Seventh Conference on Uncertainty in Artificial Intelligence, UAI 2021*, ser. Proceedings of Machine Learning Research, vol. 161, AUAI Press, 2021, pp. 493–503. [Online]. Available: `https://proceedings.mlr.press/v161/eriksson21a.html`.

[41] R. Moriconi, M. P. Deisenroth, and K. S. S. Kumar, "High-dimensional bayesian optimization using low-dimensional feature spaces," *Mach. Learn.*, vol. 109, no. 9-10, pp. 1925–1943, 2020. DOI: `10.1007/S10994-020-05899-Z`. [Online]. Available: `https://doi.org/10.1007/s10994-020-05899-z`.

[42] Y. Gur, D. Yang, F. Stalschus, and B. Reinwald, "Adaptive multi-model reinforcement learning for online database tuning," in *Proceedings of the 24th International Conference on Extending Database Technology, EDBT 2021*, OpenProceedings.org, 2021, pp. 439–444. DOI: `10.5441/002/EDBT.2021.48`. [Online]. Available: `https://doi.org/10.5441/002/edbt.2021.48`.

[43] J. Kossmann, S. Halfpap, M. Jankrift, and R. Schlosser, "Magic mirror in my hand, which is the best in the land? an experimental evaluation of index selection algorithms," *Proc. VLDB Endow.*, vol. 13, no. 12, pp. 2382–2395, Jul. 2020, ISSN: 2150-8097. DOI: `10.14778/3407790.3407832`. [Online]. Available: `https://doi.org/10.14778/3407790.3407832`.

[44] I. Mami and Z. Bellahsene, “A survey of view selection methods,” *SIGMOD Rec.*, vol. 41, no. 1, pp. 20–29, Apr. 2012, ISSN: 0163-5808. DOI: 10.1145/2206869.2206874. [Online]. Available: https://doi.org/10.1145/2206869.2206874.

[45] X. Zhou, L. Liu, W. Li, *et al.*, “Autoindex: An incremental index management system for dynamic workloads,” in *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, 2022, pp. 2196–2208. DOI: 10.1109/ICDE53745.2022.00210.

[46] S. Chaudhuri and V. R. Narasayya, “Autoadmin 'what-if' index analysis utility,” in *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '98, ACM Press, 1998, pp. 367–378. DOI: 10.1145/276304.276337. [Online]. Available: https://doi.org/10.1145/276304.276337.

[47] Y. Wu, X. Zhou, Y. Zhang, and G. Li, “Automatic index tuning: A survey,” *IEEE Trans. on Knowl. and Data Eng.*, vol. 36, no. 12, pp. 7657–7676, Dec. 2024, ISSN: 1041-4347. DOI: 10.1109/TKDE.2024.3422006. [Online]. Available: https://doi.org/10.1109/TKDE.2024.3422006.

[48] Databricks, *Databricks documentation*, https://www.databricks.com/blog/2018/07/31/processing-petabytes-of-data-in-seconds-with-databricks-delta.html, Accessed: 2025-07-01, 2025.

[49] deltalake, *Delta lake documentation*, https://delta.io/blog/delta-lake-3-2/, Accessed: 2025-07-01, 2025.

[50] P. Documentation, *Parquet documentation*, https://parquet.apache.org/, Accessed: 2025-07-01, 2025.

[51] A. Hive, *Apache hive documentation*, https://hive.apache.org/docs/latest/languagemanual-orc_31818911/, Accessed: 2025-07-01, 2025.

[52] R. Bayer, "The universal b-tree for multidimensional indexing: General concepts," in *Proceedings of the International Conference on Worldwide Computing and Its Applications*, ser. WWCA '97, Berlin, Heidelberg: Springer-Verlag, 1997, pp. 198–209, ISBN: 354063343X.

[53] A. D. Documentation, *Amazon zorder*, https://aws.amazon.com/blogs/database/z-order-indexing-for-multifaceted-queries-in-amazon-dynamodb-part-1/, Accessed: 2025-07-01, 2025.

[54] A. D. Documentation, *Amazon zorder*, https://aws.amazon.com/blogs/database/z-order-indexing-for-multifaceted-queries-in-amazon-dynamodb-part-2/, Accessed: 2025-07-01, 2025.

[55] V. Nathan, J. Ding, M. Alizadeh, and T. Kraska, "Learning multi-dimensional indexes," in *Proceedings of the 2020 ACM SIGMOD international conference on management of data*, 2020, pp. 985–1000.

[56] J. Ding, V. Nathan, M. Alizadeh, and T. Kraska, "Tsunami: A learned multi-dimensional index for correlated data and skewed workloads," *Proc. VLDB Endow.*, vol. 14, no. 2, pp. 74–86, Oct. 2020, ISSN: 2150-8097. DOI: 10.14778/3425879.3425880. [Online]. Available: https://doi.org/10.14778/3425879.3425880.

[57] J. Lao, Y. Wang, Y. Li, *et al.*, "Gptuner: A manual-reading database tuning system via gpt-guided bayesian optimization," *Proc. VLDB Endow.*, vol. 17, no. 8, pp. 1939–1952, Apr. 2024, ISSN: 2150-8097. DOI: 10.14778/3659437.3659449. [Online]. Available: https://doi.org/10.14778/3659437.3659449.

[58] M. Pöss, R. O. Nambiar, and D. Walrath, "Why you should run TPC-DS: A workload analysis," in *Proceedings of the 33rd International Conference on Very Large Data Bases, VLDB 2007*, ACM, 2007, pp. 1138–1149. [Online]. Available: http://www.vldb.org/conf/2007/papers/industrial/p1138-poess.pdf.

[59] IBM, *Ibm documentation*, https://www.ibm.com/new/announcements/ibm-granite-4-0-tiny-preview-sneak-peek, Accessed: 2025-07-01, 2025.

[60] I. D. Documentation, *Db2 co database setup*, https://www.ibm.com/docs/en/db2/12.1.0?topic=to-creating-setting-up-your-database-configuration-analytic-workloads, Accessed: 2021-10-04, 2022.