Guiding Expert Data Systems Tuning with Explainable AI

Andrew Chai

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

April 2025

Abstract

Modern database systems, such as IBM Db2, rely on cost-based optimizers to improve workload performance. However, their decision-making processes are difficult to interpret. Tuning them for specific workloads remains challenging due to their complexity, numerous configuration options, and interaction with unique workload characteristics. Additionally, database systems increasingly rely on black-box machine learning models within the optimizer and automatic tuning tools. These black-box models lack interpretability, hindering expert trust and debugging. We propose GEX, a system that provides interpretable insights into database optimizer behavior using explainable AI techniques. We adapt XAI techniques for generating perturbation-based saliency maps from surrogate models to the domain of SQL queries. With GEX we propose a framework for how saliency scores can be used to guide experts in system tuning tasks such as statistical view creation, configuration parameter adjustment, and query rewrite. We demonstrate the ability of GEX to capture and communicate optimizer behaviour through experimental evaluation in these tasks using the TPC-DS benchmark and IBM Db2.

Dedication

I dedicate this thesis to my grandfather. Your strength has always inspired me to overcome any challenge and be the best person I can be. I love you Gong Gong you will always be with me.

I would like to express my gratitude to my supervisors. Dr. Jarek Szlichta for introducing me to research, databases, and explainable AI. His guidance has helped me find the perfect fit for my academic interests. His direction has been invaluable in guiding me toward my goals. Dr. Parke Godfrey for his direction and advice. His enthusiasm for interesting problems has been infectious and is always motivating.

I am thankful to my IBM colleagues, Vincent Corvinelli, Calisto Zuzarte, Connor Henderson, Brandon Frendo, Nicholas Ostan, and Daniel Zilio for their valuable expertise and collaboration which helped me overcome technical challenges, and motivated this work. Vincent Corvinelli and Calisto Zuzarte, in particular, for contributing significantly to my understanding of database management systems.

Finally, I would like to give thanks to my loving wife Dakota. Her unwavering support and sacrifice through the highs and lows of every step of this journey has given us the opportunity to build the life we dream of and given me the chance to do work that I am passionate about.

Acknowledgements

I would like to acknowledge Alexander Bianchi of York University for his contributions towards `GEX`. We collaborated on the experimental evaluation of `GEX` using `Db2une` for the automatic knob tuning performance evaluation, and the experimental evaluation of the system for query rewrite determination.

Contents

List of Figures

List of Tables

List of Abbreviations and Symbols

ML: Machine Learning

AI: Artificial Intelligence

XAI: Explainable Artificial Intelligence

GEX: Guiding Expert tuning with eXplainable AI system

TBSCAN: Table Scan

HSJOIN: Hash Join

GRPBY: Group By

IBM: International Business Machines Corporation

DBA Database Administrator

OLAP: Online Analytical Processing

QEP: Query Execution Plan

SQL: Structured Query Language

TPC-DS: Transaction Processing Performance Council Decision Support Benchmark

Q_x: Query x of the TPC-DS workload

Saliency-Inf Saliency-Informed

FF: Filter Factor

Chapter 1

Introduction

1.1 Motivation

Modern database systems, including `IBM Db2`, rely on sophisticated cost-based optimizers[1] to transform SQL queries into efficient query execution plans (QEPs), ensuring optimal workload performance. Query optimizers use complex algorithms, often enhanced with machine learning [2]–[4], to estimate costs and select efficient execution paths. However, the increasing complexity of these systems presents challenges for expert database administrators. This complexity limits their ability to interpret system behavior, diagnose improper handling of complex workload characteristics, and subsequently tune them for better performance.

Data systems expose many configuration options for experts to improve system performance, including the definition of improved statistics collection, optimization of resource allocation, and manual adjustments to optimizer behavior. Declaring *statistical views* [5] can improve cardinality estimation by collecting statistics for important columns and portions of queries, such as joins, where complex relationships

such as skews and correlation may not be correctly captured by standard statistics collection. Improved cardinality estimates for these relationships result in better cost estimates, plan generation, and execution time. Adjusting *configuration parameters* ("knobs") [6] tailors system performance to workload requirements by customizing resource allocation, such as main memory, and execution behavior, such as degree of query parallelism [7]. Applying *query rewrites* can simplify and restructure queries for more efficient evaluation [8]. Rewrites can be prompted by registry variables [9], making fine-grained manual adjustments to the optimizer's behavior. Due to the numerous interdependent strategies available, the complexity of the optimizer, and the unique needs of each workload, tuning is a challenging task. This necessitates systems that can provide actionable insight to assist in these tasks and the need for automated systems that can perform them with minimal human intervention.

Automated systems have been introduced that can perform knob tuning [10]–[15]. The state of the art for these systems have shown that they are competitive with, or even surpass, human experts. However, their use of opaque machine-learning models [16], [17] limits their utility, as so-called *black-box* approaches, such as deep neural networks, are not directly human-interpretable. While these automatic tuning systems are potentially very effective, their lack of *interpretability* makes it difficult for experts to trust their recommendations and diagnose performance regressions. For these reasons, research on improving insight into these models is a high-priority goal for our industry collaborators at `IBM`.

The challenges with the interpretability of black-box models are not unique to data systems. Considerable work has been conducted in the field of *eXplainable Artificial Intelligence* (XAI), with the aim of addressing these problems. XAI techniques, such

as surrogate models, have been used to generate saliency maps that highlight important features [18]–[21], providing interpretable insights into complex model behaviors. These techniques have proven effective in image processing [22], as well as in tabular [23] and text analysis [24]. Given the increased prevalence of machine learning in automated data systems and within the optimizer along with the existing complexity of database optimizers and workloads, there is a potential for these techniques to be used to great effect. Applying these methods to the components of database systems could explain how the features of SQL queries affect their decisions, informing experts about system behavior, enabling better informed tuning adjustments. Achieving this requires novel adaptions of the XAI techniques to accommodate the unique features of SQL queries and query plans.

1.2 Contributions

We have designed and developed a novel approach for using explainable AI to explain data systems: `GEX`, *Guiding Expert tuning with eXplainable AI* [25]. Our system `GEX` uses interpretable surrogate models to generates saliency maps for SQL queries and their execution plans, identifying the components of queries that have the greatest influence on cost estimations. This offers experts actionable insights into the behaviour of black-box components of data systems, informing tuning strategies. This system has resulted in one paper publication that has been accepted to appear in the upcoming 41st IEEE Conference on Data Engineering [ICDE]. This thesis reprints portions and figures from that publication with the permission of IEEE [25].

In Section 2, we provide background for saliency score generation and discuss the challenges of system tuning. We provide a system design description in Section 3.1

and evaluate the effectiveness of the system in Section 4. Our main contributions are as follows.

1. **Workflow Design and System Integration.** (Section 3) `GEX` proposes a novel workflow for database tuning, integrating XAI techniques with the `IBM Db2` optimizer. It generates and analyzes query saliency maps, producing interpretable insights that guide both manual and automated optimization strategies.

   (a) **Query Saliency Maps.** (Section 3.2)

   We introduce a method, the first of its kind, for generating query-specific saliency maps using interpretable surrogate models. This approach models the effect of changes in cardinality estimates for query predicates on the optimizer's cost estimates by perturbing *filter factors*—which represent the proportion of rows satisfying a predicate—and observing their impact on optimizer decisions. These saliency maps provide transparency into how specific query elements influence the cost to evaluate the query.

   (b) **XAI-Driven Data-System Tuning.** (Section 3.4)

   We propose a framework that integrates XAI techniques, specifically our query saliency maps from above, into database-system tuning. Our framework equips experts with actionable insights into query performance optimization. This enables experts to trace and debug optimizer behavior by identifying high-impact query and execution plan components, facilitating informed decisions for downstream tuning tasks. We provide a `GEX` informed processed for: *statistical view* definition to collect relevant workload statistics, *query rewrites* by identifying salient subplans, and configuration *"knob" tuning* by quantifying changes in query operator cost relationships.

2. **Experimental Evaluation.** (Section 4)

   We validate GEX on the TPC-DS benchmark workload [26], demonstrating its significant performance improvements, in both manual and automated tuning scenarios.

   (a) **Statistical Views.** GEX guides the selection of statistical views to improve cardinality estimates, achieving up to 48% improvement in query run times and delivering recommendations that surpass those of IBM experts with over 20 years of experience.

   (b) **Knob Tuning.** GEX uses insights derived from saliency maps to inform adjustments to database configuration parameters, including those used by automated tuning systems, reducing query execution by up to 42% compared to baseline configurations.

   (c) **Query Rewrites.** GEX highlights optimization opportunities, identifying worthwhile query rewrites, improving query run times by up to 66%.

   These results show the effectiveness of GEX in tuning data systems by providing interpretable insights into the cost function of the IBM Db2 optimizer, enhancing the understanding understanding of experts and assisting in the debugging of optimizer decisions.

We include a chapter on related work in explainable AI in Section 5, and conclude with a summary and future work in Section 6.

Chapter 2

Background

2.1 Explainable AI Saliency

One of the primary modes of providing an explanation using explainable AI is to generate *saliency scores* [27]. Saliency scores assign values to the features of an input indicating their relevance to the decisions of a system [27]. Numerous methods have been developed for the computation of saliency scores in differing domains of model input data types. Methods can be divided into two main approaches *gradient-based*, which take advantage of the gradients of the model they explain to propagate relevance scores back to input features [28], [29], and *perturbation-based*, which make small changes or *perturbations* to input features, measure changes in model output, and then compute feature relevance [18], [19]. Given that our method is applied to a DBMS cost-optimizer which does not generate gradients with which to compute scores, we focus on perturbations methods in this work. We further discuss these methods as well as additional modes of explanation in related work (Section 5).

Existing methods such as SHAP [19] and LIME [18] have well-defined perturbation-

based approaches to generate saliency scores for text, image, and tabular data. To create data with which to compute saliency scores, these approaches: take a given input data point for which the behaviour of the model is to be explained, perform perturbations to that data point, measure model output after running inference on the perturbed data, and use the changes in model output to compute the relevance each feature. For text data, perturbations are most often performed by masking or removing individual words or tokens from the text. Images are perturbed either pixel- or superpixel-wise with combinations of sections of the image blacked out or greyed out. Tabular data are perturbed attribute-wise by sampling random attributes from the dataset (in the case of SHAP) or sampling from a computed distribution of an attribute (in the case of LIME) [27]. Through repeated perturbations and sampling, data about the decisions the model makes with respect to the features of the original sample are captured.

SHAP uses these data, along with a custom kernel weighting function which takes into account the distance of a perturbation, to train a linear model that estimates *Shapley Values* [19]. Shapley values are an approach rooted in game theory to assign the contribution of individuals to a final outcome [27], [30]. LIME uses the data sampled, weighted by the distance from the original, to create a *local surrogate model*. *Local* in this case refers to the *local* region of model decisions based on the initial sample which is to be explained and similar samples generated by perturbation [18]. A *Surrogate model*, in this case of LIME, usually refers to a linear model or other interpretable model used to represent the local decision space. Linear models are used because they are inherently interpretable as they consist of a weighted linear sum of input features for which the weights can represent the significance of a feature.

However, they are also considerably less expressive than more complex models such as deep neural networks that LIME can be used to explain. This is the reason why the surrogate model is constrained to explaining only the local decisions of the model [18]. We describe how our system adapts the concept of input perturbation and local surrogate modeling in the system overview (Section 3).

2.2 System Tuning Challenges

As mentioned in motivation, data system tuning remains a challenging task for even expert database administrators. The challenge of this task stems from many sources: the increasing complexity of the cost-based query optimizer; schema data characteristics such as correlations and skew; query-workload-specific challenges such as complex analytical queries; and numerous available configuration options such as statistical views, configuration parameter knobs, and optimizer behaviour altering registry variables. Even selecting a single scope of optimization such as knob tuning can be challenging, as systems can expose hundreds of configuration knobs that present a search space that cannot be exhaustively searched [7], especially considering potentially long query run times with poor knob configurations. Each of these complexities interacts with each other, further complicating the task.

To illustrate this challenge and the expert knowledge required, we discuss some of the considerations an internal IBM expert database administrator, DBA, can make for the configuration of `IBM Db2` using Query 1 ($\mathbf{Q}_1$) of the TPC-DS decision support benchmark as an example [26]. These considerations were provided through numerous discussions with our collaborators, industry experts Calisto Zuzarte and Vincent Corvinelli at IBM, who have done considerable work directly on the `IBM`

Db2 optimizer. IBM Db2 uses a cost-based optimizer to evaluate potential query execution plans and select efficient access paths. These cost estimates represent the estimated cpu usage and disk I/O required for a given operator. These estimates are computed considering factors such as the cardinality of the data to which the operation is applied, as determined by the statistics collected by the system, and the resources available for the operation as defined by the system's tuning (settings of the configuration parameter knobs such as memory allocation knobs *sort heap* and *buffer pool*). The scope of operations that can be considered can be controlled using knobs that affect optimizer behaviour such as *optimization level* and registry variables that prompt the optimizer to consider specific rewrites.

The correct configuration of IBM Db2 for this example requires significant expert knowledge of the system, the workload, and their interactions. Knowing that the TPC-DS workload has a star schema and consists of complex analytical queries, an expert may choose to load the data in column-store format as that generally results in the best performance for analytical queries. The expert will then need to rely on knowledge of optimizer behaviour in this environment to select appropriate knobs to configure from dozens of possibilities and how these should be configured for the given data scale (in our case 100GB).

For this workload, the expert may prioritize tuning the sortheap knob, given their knowledge that the optimizer frequently selects hash-joins as the join type for column store and that hash-join buffers are stored in the sortheap [31]. When the sortheap is insufficient for the cardinality of the join, spilling to disk will occur, incurring additional I/O costs. If the optimizer accurately predicts spilling or selects otherwise more expensive plans to avoid it, the additional cost will be reflected in the cost

estimate of the query execution plan. This can be shown by sampling execution plans generated with different sortheap configurations. Figure 3.3 shows the estimated cost of Query 1 at varying sortheap and buffer pool settings.

If the optimizer does not accurately predict the number of pages needed, spilling can occur at runtime, causing unexpected degradation to system performance. The cardinality estimation of the hash-join, and subsequently the number of pages it will require depends on the system accurately collecting statistics about the join. In some cases, standard statistics collection is insufficient to capture complex correlations and skews for relationships within a schema. To improve statistics collection, *statistical views* can be defined to instruct the database to collect additional statistics about specific relationships. An expert can apply statistical views to correct these errors, given they have system-specific knowledge of the kinds of relationships that are not accurately captured by default and workload-specific knowledge of the relevant relationships in the workload.

In the case of TPC-DS, IBM experts have identified several relationships which may cause cardinality estimation errors. Query 1 contains one of these in the join between the `store_returns` and `date_dim` table. This expert knowledge of the workload, or insight gained from `GEX` as described in Section 3.4, can inform the creation of a targeted statistical view, shown in Fig. 3.4. Figure 3.5 shows the updated cost estimates for query 1 after creating the view. The application of the view corrected for an underestimation in cardinality, resulting in higher and more accurate cost estimates, particularly at lower knob settings. This is because the `date_dim` table is "overloaded": there is a significantly larger range of dates in it than the range of values the predicate applied to the join addresses, causing the optimizer to underestimate

the cardinality.

The knowledge required and the challenge posed by tuning even a single knob for a single query workload are repeated for each configuration option for every unique workload. Given the time-consuming nature of manual tuning, automatic tuning systems have been developed [10]–[15]. However, as previously mentioned in the motivation, these methods are not human interpretable. This poses a challenge for our IBM experts, because, regardless of whether the tuning is manual or automatic, there is the potential to cause performance regressions by recommending worse configurations. Due to the black-box nature of these systems, regression debugging still requires laborious manual investigation and tuning.

In the system overview (Section 3), we address how `GEX` can be used to inform the manual definition of statistical views, configuration of knob settings, and assist with the analysis of automatic tuning systems. We evaluate the results of this informed process in the experimental evaluation (Section 4).

Chapter 3

System Overview

3.1 System Design Overview

`GEX` leverages XAI techniques to provide interpretable insights into database optimizer behavior, enabling experts to make informed and explainable tuning decisions. The system captures optimizer behavior through the use of an explainable linear surrogate model to create saliency scores that are mapped to features of the query and corresponding query execution plan. Through this modeling, we provide insight into the optimizer's cost model and decision-making, providing actionable tuning guidance. The workflow design is illustrated in Fig. 3.1 and is comprised by the following steps.

- **Workload Input.** SQL queries are passed to the database optimizer (Step 1) and a parsing module [32] (Step 2).
- **Filter Factor Extraction and Perturbation.** The optimizer outputs predicate filter factors (Step 3). These are paired with tagged predicates from the SQL parser (Step 4) and perturbed slightly to create modified queries (Step 5).

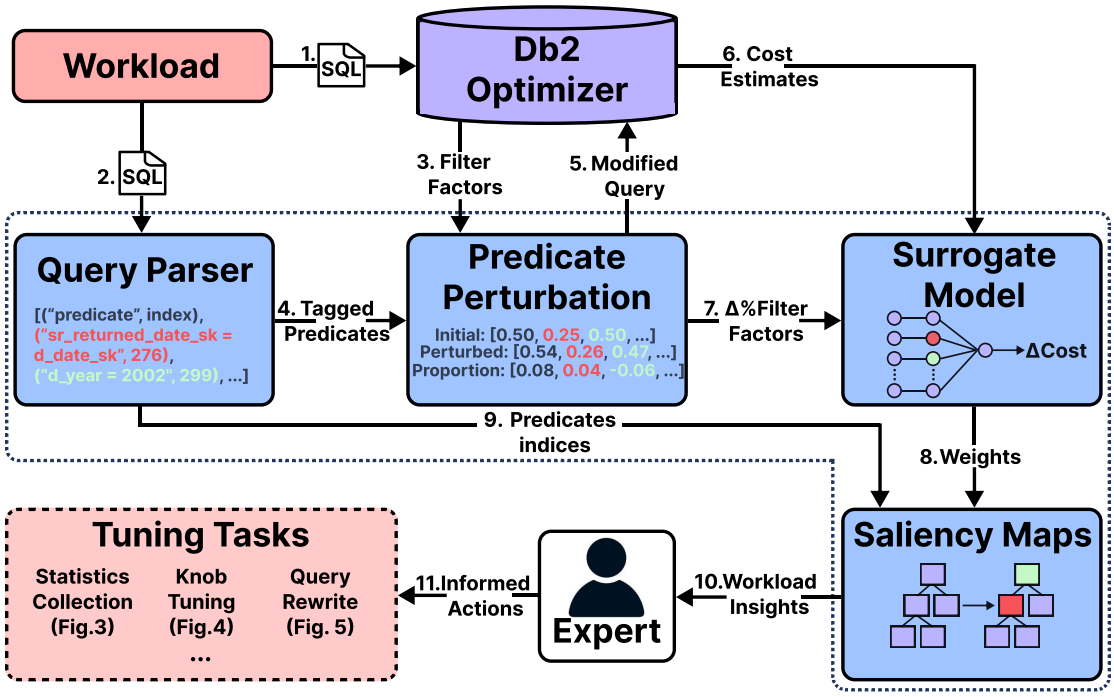

Figure 3.1: GEX system workflow design. © 2025 IEEE

- **Optimizer Cost Estimation.** Each modified query is passed to the optimizer, and the resulting updated cost estimates are gathered (Step 6). This forms a dataset of filter factor perturbations and their corresponding costs (Step 7).
- **Surrogate Model Training.** The collected dataset is used to train a linear surrogate model that models the relationship between filter-factor changes and cost estimates. By extracting the weights of the surrogate model (Step 8) and matching them to the predicates (Step 9), we construct saliency maps for the SQL query and its execution plan.
- **Workload Insights and Actions.** Saliency maps highlight influential predicates, providing interpretable insights into optimizer behavior to experts (Step 10). These insights guide downstream tuning actions, such as creating statistical views, adjusting knobs, or rewriting queries (Step 11).

By perturbing filter factors and analyzing the optimizer's cost responses, `GEX` generates query saliency maps that reveal the impact of individual query components. This interpretable framework enables targeted tuning strategies, helping to close the gap between the complexity of modern optimizers and the transparency required by experts. Detailed explanation of the process and the intuition behind it are discussed next.

3.2 Surrogate Modeling

3.2.1 SQL Query Perturbation

`GEX` contributes a novel adaptation of linear surrogate modeling for SQL queries to approximate the optimizer's cost decision process, enabling the creation of *query saliency maps* that highlights the most influential predicates in a SQL query and its execution plan.

As discussed in the background (Section 2), existing XAI methods SHAP [19] and LIME [18] use well-defined perturbation strategies to sample modified versions of text, image, and tabular data [27]. However, SQL queries, which are the input data for query optimizers and tuning systems such as [10], [11] exhibit unique properties that do not align with these methodologies. For instance, text perturbations typically remove combinations of words or tokens, treating the text as a bag of words. While this breaks the syntax of the text, this may be acceptable in the case of identifying the words which most contribute to a classifier's confidence [24]. However, this approach would frequently break SQL syntax, rendering the perturbed SQL potentially unusable for optimizers, making it inapplicable to our use case. Similarly, image

perturbations mask specific regions, typically by greying-out sections of the image, effectively removing those regions from the model's consideration [27]. For SQL this could correspond to removing part of the query. However, for complex analytical queries with a tree-like logical structure, later operations in the query and their associated operations can be dependent on the results of earlier operations. By removing parts of a query, rewrites would be required which result in significant changes in query semantics and costs. For example, a join which appears higher up in the query plan may have one of its join predicate columns introduced by an earlier join. In this case, if a perturbation removed the earlier join, it would also necessitate the removal of the later join or significant additional rewrites. This poses a challenge for the definition of semantically valid perturbations of a query. Tabular data perturbations typically involve sampling replacement values for features from other entries in a dataset. However, this method inherently relies on an existing set of input data with the same features. To perform perturbations-based modeling for SQL queries, we must identify a feature that can be meaningfully perturbed without too significantly altering semantics or breaking the syntax of the query.

Cost-based optimizers calculate an estimated *filter factor*, also called a *reduction factor*, $f_i \in [0, 1]$, for each predicate i in an SQL query. These filter factors are quantified as *selectivity estimates*, representing the predicted proportion of rows satisfying a predicate. We consider these filter factors as a key implicit feature of a SQL query when working with systems that use cost-based optimizers. Importantly, these filter-factors are significant to the cardinality and therefore cost estimates of query operations. These can be modified without changing query semantics, and are common features due to the extensive use of predicates in analytical query workloads.

We extend the idea of perturbation based saliency to SQL queries by proposing a novel approach. `GEX` applies small, localized perturbations to filter factors. This preserves the query's semantics by using the unique characteristics of SQL queries. Using the `IBM Db2 EXPLAIN` command and `SELECTIVITY` keyword statements [33], `GEX` inserts the perturbed filter factors into the query as directives. These directives then modify the optimizer's planning by overriding the filter factors it would have computed.

For a given query Q, `GEX`

1. *extracts* the original filter factors $F_Q = \{f_1, f_2, \ldots, f_n\}$ using `EXPLAIN`,
2. *perturbs* each filter factor f_i to f_i' to form F_Q', and simulating small changes in predicate selectivity by sampling a new value from a normal distribution $\mathcal{N}(\mu = f_i,\, \sigma^2 = f_i * 0.05)$,
3. *injects* the perturbed filter factors into the optimizer using the `SELECTIVITY` statement to recompute query cost C_Q'.

The proportional change in filter factors (3.1), and corresponding change in the optimizer's cost estimate (3.2) are defined as:

$$\tilde{f}_i = f_i'/f_i - 1 \qquad (3.1)$$

$$\tilde{C}_Q = C_Q' - C_Q \qquad (3.2)$$

in which C_Q is the original cost and C_Q' is the new cost estimated after reinserting the perturbed filter factors. These proportional changes form a dataset of feature-response pairs, $(\tilde{F}_Q, \tilde{C}_Q)$, to train the surrogate model. The key intuition behind this process is that changes in filter factors are analogous to changes in the cardinality of rows passing through that predicate. By perturbing this cardinality, we can change

the number of rows and associated costs of operators and portions of a query. This allows us to capture the connection between the cost estimate and cardinality for the predicate and subsequent portions of the query.

Perturbations, as shown in Step 2, are defined using a normal distribution with the variance being a percentage of the original predicate. This was selected because, while all filter factors fall in a range of $[0, 1]$, predicates within a query may be many different orders of magnitude in size. A fixed offset or range of offsets for filter factor perturbation would disproportionally affect predicates with varying scaled values. Therefore, to perturb the predicates *locally*, and on average, proportionally across many samples, we scale the perturbation factor to have a variance within a percentage of the original. Considerations for the n number of samples required and the $\sigma^2 = f_i * 0.05$ variance of the samples for a given filter factors are made with respect to the fit of the surrogate model and are discussed in the following subsection.

Due to the extensive rewrites the database optimizer performs, matching the filter factors to their original predicate in Step 3 is not trivial. `IBM Db2` reports filter factors with respect to the predicates within the optimized SQL, which have often had portions of the predicate columns rewritten; for example, with the table names being replaced with aliases. Additionally, predicates may be restructured to varying extent; for example, a greater or equal to operation may be converted to a less than operation with the order of the predicates swapped. This means that simple string matching of predicate text is not usable. Fig. 3.2 shows an example predicate before and after query rewrites, note the change in case, aliasing of the source table, and rearrangement of the equality.

For these reasons, `GEX` implements a two-stage approach to predicate matching.

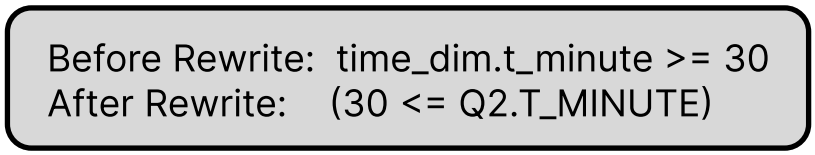

Figure 3.2: Predicate from TPC-DS $\mathbf{Q}_{96}$ before and after query rewrites

With the assumption that if a predicate has the same optimizer computed filter factor inserted using the selectivity clause, the resulting query plan will have the same cost. First, we consider iterating through all possible filter factors returned by the optimizer. However, with the potential for many predicates to exist within an analytical query, this is expensive, with a $\mathcal{O}(n^2)$ cost for n predicates within a query. Many rewritten predicates are similar to their originals, retaining column names or literals. Therefore, for our first matching stage, we apply fuzzy string matching to create an ordering for the list of predicates to test. This resulted in 1209 out of the 1282 predicates from the TPC-DS benchmark queries that we tested matching on the first try. However, even with this approach, in the best case, 62 predicates were still not matched. This is not unexpected given that the query rewrite engine may elect to split or combine predicates, resulting in the rewritten query missing a direct rewrite of the predicate and its associated filter factor. It would still be of value to assign a filter factor to these predicates.

Addressing this, we implement a secondary stage to our filter factor matching strategy. Given that the filter factors are bound to the range $[0, 1]$ and assuming that the cost with respect to a predicate is monotonic. We perform a binary-search approach by injecting a placeholder filter factor of 0.5 and then repeatedly measure if the cost is higher, lower, or equal to the baseline for the query and update the placeholder filter factor accordingly. This approach can be unsuccessful in the case

where the cost of the query is not monotonic with respect to the predicate's filter factor; this can occur in cases where the optimizer selects a different execution plan due to changed cardinality estimates. Through this binary matching, which is applied after fuzzy matching, we match an additional 53 predicates, leaving only 20 predicates out of 1282 remaining unmatched. We consider this acceptable for our purposes as 98.4% of the predicates that we target are matched with a filter factor and can be perturbed by our system.

3.2.2 Linear Surrogate Model

`GEX` uses a linear surrogate model to approximate the relationship between filter factor perturbations and cost changes:

$$\hat{C}_Q = \sum_{i=1}^{n} \tilde{f}_i \theta_i + \theta_0 \tag{3.3}$$

The learned weight θ_i for predicate i, represents its sensitivity to cost changes, and θ_0 is the bias term, which accounts for constant effects not tied to specific predicates.

We selected a linear model as our surrogate for the following reasons.

- **Interpretability.** Linear models are inherently interpretable, allowing the influence of the change in each predicate's filter factor to be directly quantified by its weight θ_i. This allows the weights to be used to create saliency scores with a clear connection to the relationship being modeled. In this case, the relationship between changes in cost and cardinality for portions of a query. In exchange for the interpretability gained due to the simplicity of the model, we trade-off model expressiveness, which leads to the next point.

- **Locality Constraints.** By constraining the perturbation filter factors to a small percentage of the original before injecting them using the `SELECTIVITY` statement, we can constrain the space of the optimizer's cost model to the query to a set of query execution plans with the same semantics and similar cardinality. This reduced the range and complexity of the behaviour to be explained, allowing for a linear model with reasonable fit.
- **Efficiency**. Linear models are computationally lightweight and can be trained on modestly sized datasets generated by the perturbation process of `GEX`. This is necessary as repeated sampling of query execution plans from the optimizer incurs a computational cost as plans must be generated for each new set of filter factors.

The surrogate model is trained by minimizing the following objective function:

$$\mathscr{L}(\theta) = \sum_{j=1}^{m} \left(\tilde{C}_Q^{(j)} - \sum_{i=1}^{n} \tilde{f}_i^{(j)} \theta_i + \theta_0 \right)^2 \tag{3.4}$$

Here, m is the number of perturbed queries in the dataset, $\tilde{C}_Q^{(j)}$ is the observed proportional change in cost for the j-th perturbed query, and $\tilde{f}_i^{(j)}$ is the proportional change in the filter factor for the predicate i in the j-th perturbed query. By minimizing this loss, the surrogate model assigns weights θ_i that reflect the impact of each predicate on the optimizer's cost estimates.

Described in Section 3.2.1, our query perturbation step has two primary parameters; the number of samples and the percent of the original filter factors that we define as the variance of the normal distribution sampling function. Both of these parameters were selected with the aim of creating linear models with a good coefficient of determination score r^2. We found that $n = 1000$ samples and a variance

of $\sigma^2 = f_i * 0.05$ tended to produce models with high r^2 scores, of upwards of 0.95. However, when analyzing the residual errors of the models, we found that this did not always mean that the models were a good fit. This suggests that extending the models with interaction terms or higher-order terms may be of value. In future work, Section 6.2, we discuss the possibility of extending this work to use more sophisticated models with additional terms. In experimental evaluation, Section 4 we find that the linear models with our selected parameters are sufficient to generate saliency scores for guiding tuning decisions, and that our scores align with expert intuition.

3.3 Saliency Map Creation

To compute the saliency scores, the weights θ_i learned by the surrogate model are normalized in a range of [-1,1]. They are then inserted into the original query beside each of the original predicates to produce *query saliency maps*, visually indicating the impact of each predicate on the optimizer's cost estimates. Query saliency maps are extended to *QEP saliency maps* by matching predicates and weights with their respective operators in the plan. High-weight predicates are flagged as critical for tuning, guiding experts to effective optimization decisions.

Fig. 3.3 illustrates a query saliency map for TPC-DS query 1. The predicates are color-coded based on their normalized weights θ_i. Predicates a colour coded on a green-to-red gradient based on score. Predicates with higher scores, shown in darker shades of red, indicate a stronger impact on the optimizer's cost estimates; predicates with lower scores are displayed as green, reflecting minimal influence. These high-score predicates guide experts to focus their efforts on the most critical areas of the query, enabling more effective tuning actions, such as improving cardinality estimates

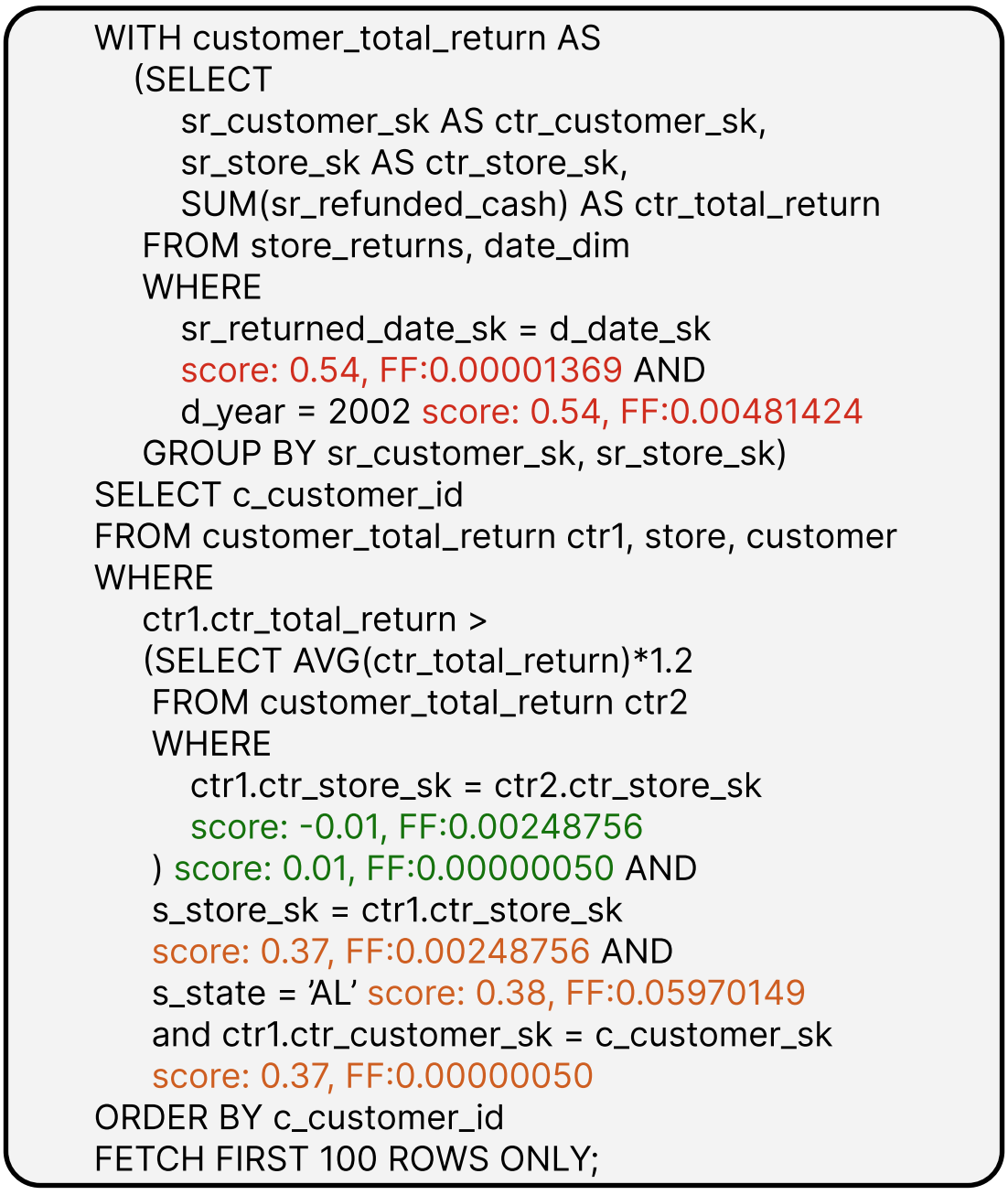

Figure 3.3: Saliency map for TPC-DS Query 1. © 2025 IEEE

through statistical views. For example, in Fig. 3.3 the predicates that define the join conditions between `store_returns` and `date_dim` have the highest score and are highlighted in dark red. This means that they are predicates in this query for which the total cost of the query is most sensitive to changes in cardinality. This aligns well with the discussion of tuning `IBM Db2` for this query in Section 2.2. Given the sensitivity of these predicates and their relevance to the total cost of this query, it would be reasonable to be guided by `GEX` scores to consider defining a statistical view to guarantee accurate statistical modeling of it. We further discuss `GEX`'s application for guided system tuning, including this example in the following section.

3.4 Guided Tuning

Advanced database systems, such as `IBM Db2`, offer a variety of tuning mechanisms to optimize system performance. These include defining statistical views, adjusting the configuration parameters (knobs), and rewriting queries. There is no universal tuning solution for all workloads, due to variations in schema, data distribution, and query characteristics [7], [10]. Manual tuning, while effective in expert hands, is labor intensive, and requires deep domain expertise. In contrast, modern automated tuning systems often rely on opaque, black-box machine learning models. These lack interpretability, making it challenging for experts to verify, debug and trust their recommendations.

`GEX` provides a hybrid tuning approach that combines the automation of explainable AI with expert-driven strategies. By revealing the optimizer's underlying behavior, `GEX` enables informed decisions through query saliency analysis, task prioritization, and guided optimization. This unified framework bridges the gap between

manual and automated tuning, ensuring transparency.

1. **Saliency Analysis.** `GEX` generates query saliency maps that assign saliency scores to individual query predicates. These provide experts with a clear, visual representation of the query parts that most significantly impact the optimizer's decisions.
2. **Task Prioritization.** Using the saliency maps, queries are ranked by feature sensitivity, allowing experts to focus on the highest impact workload components. This prioritization ensures tuning efforts target areas with the greatest potential for performance improvement, whether through statistical view creation, knob adjustments, or query rewrites.
3. **Guided Optimization.** Experts use the saliency maps and rankings to guide both manual and automated tuning efforts. In manual tuning, experts analyze the saliency maps of the highest-ranked queries to make adjustments based on the highlighted areas. In automated tuning, these insights serve to validate, through expert analysis, the recommendations of black-box systems, ensuring that proposed configurations align effectively with the workload's specific characteristics.

We next describe its application to specific tuning scenarios.

Statistical View Selection. Statistical views enhance the accuracy of query cost estimates by generating detailed statistics for key columns and predicates [34]. In `IBM Db2`, statistical views are defined via SQL statements and populated with the relevant statistics through the `RUNSTATS` statement. Accurate cardinality estimations are critical for cost-based optimizers, as estimation errors can propagate, leading to suboptimal plan selections and degraded query performance [1], [35].

However, selecting the optimal set of statistical views is a non-trivial task. A naive approach, such as creating views for every possible local or join predicate in a

```
CREATE VIEW view_q1 AS (
SELECT d_year, sr_customer_sk, sr_store_sk
FROM store_returns, date_dim
WHERE sr_returned_date_sk = d_date_sk);
```

Figure 3.4: Statistical view for TPC-DS $\mathbf{Q}_1$. © 2025 IEEE

workload, is computationally prohibitive and costly to maintain. GEX addresses these challenge by using saliency maps to guide statistical view selection. These identify query components with the greatest influence on optimizer cost estimates, allowing experts to focus on predicates where cardinality errors would most significantly affect performance.

In automated tuning scenarios, GEX enhances its utility by enabling experts to validate view recommendations generated by black-box systems [36], [37]. Automatic view recommendation systems generate candidate views based on workload patterns. GEX assist in analye these recommendations, highlighting the views that align with its saliency insights likely to have the most impact. This ensures that automated systems produce actionable and trustworthy outputs, bridging the gap between automation and expert validation.

For example, in the TPC-DS workload, GEX identified critical join predicates for the `store_returns` and `date_dim` join in $\mathbf{Q}_1$, as presented in Fig. 3.3. This insight aligns with our expert example in 2, and can inform the creation of a targeted statistical view, shown in Fig. 3.4. We evaluate the improvement of query performance using statistical views informed by GEX in Section 4.1.

Additionally, for the purpose of evaluating GEX's ability to capture information relevant to statistical view recommendation, we also develop a system to use saliency

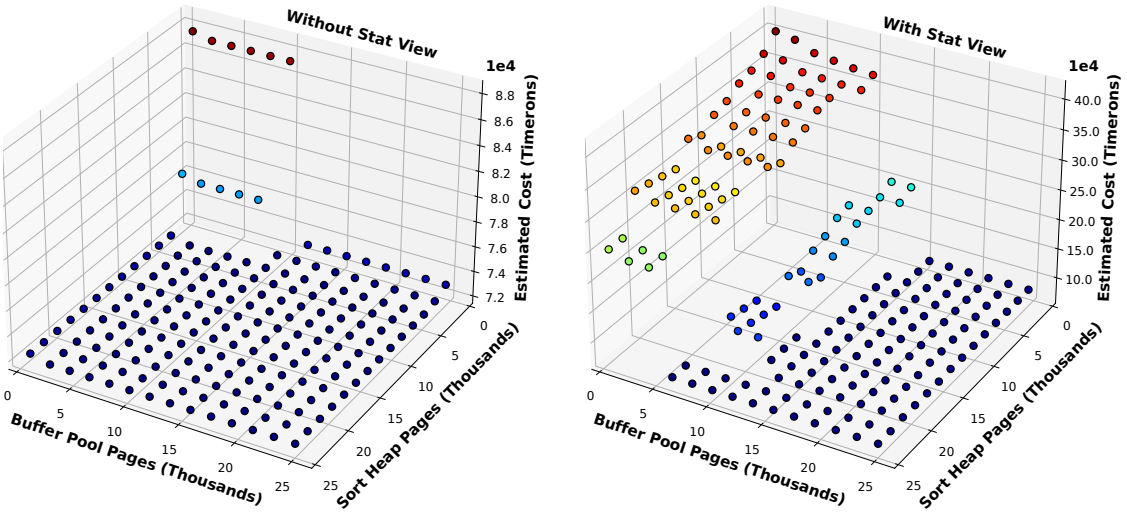

Figure 3.5: Cost estimates of TPC-DS $\mathbf{Q}_1$ at varied settings. © 2025 IEEE

scores generated by `GEX` to automatically recommend statistical view candidates. For a given workload of queries:

1. we *parse* each query into its abstract syntax tree representation;
2. *resolve* each predicate including corresponding columns and gex scores to their base tables;
3. then for the workload we *aggregate* all possible two-way join candidates including corresponding predicates and scores which appear in the workload;
4. *score* the candidates by taking a weighted sum of their gex scores multiplied by query runtime; and
5. then finally *generate* statistical view definition SQL for the top-k candidates.

We evaluate the performance of automatically defined statistical views in Section 4.1.1.

Knob Tuning Recommendation. Configuration knobs improve performance by allocating resources and specifying optimizer behavior [7]. Fig 3.5 shows the estimated cost of $\mathbf{Q}_1$ at varied memory knob settings with and without applying the statistical view presented in Fig. 3.4. Note the change in the cost surface. The application of the view corrected for an underestimation in cardinality, resulting in higher, more accurate cost estimates, particularly at lower knob settings. This view was reached independently using `GEX` scores however, as previously mentioned, for the TPC-DS workload this is a known candidate to IBM experts. This is because the `date_dim` table is "overloaded": there is a significantly larger range of dates in it than the range of values the predicate addresses, causing the optimizer to underestimate the cardinality. Although this plot visualizes only two dimensions of the settings, the variation in cost estimates will extend across the full range of available knobs.

`GEX` improves cost estimates with statistical views and, by extension, automatic knob tuning (Section 4.2). It also contributes directly to the task by providing an additional tool for analyzing the optimizer's processing of a query at different knob settings. During the tuning of knobs—whether performed manually or using automatic tools [10]–[13]—the impact of the new settings can be evaluated by re-computing saliency maps. These maps reflect changes in predicate sensitivity, providing insight into how knob adjustments influence cost estimates.

Shown in Fig. 3.6, the scores for the predicates in $\mathbf{Q}_1$ change when the sort heap knob allocation is increased. The absolute values of the predicate's associated weights in our linear model decrease. This informs us that the cost of query operations becomes less sensitive to changes in cardinality with more of this resource. Due to the fact that our scores are normalized, the scores shift toward the remain-

(a) TPC-DS $\mathbf{Q}_1$ with predicates labeled as (Px)

```
WITH customer_total_return AS
   (SELECT
      sr_customer_sk AS ctr_customer_sk,
      sr_store_sk AS ctr_store_sk,
      SUM(sr_refunded_cash) AS ctr_total_return
    FROM store_returns, date_dim
    WHERE
      sr_returned_date_sk = d_date_sk (P1)
      d_year = 2002 (P2)
    GROUP BY sr_customer_sk, sr_store_sk)
SELECT c_customer_id
FROM customer_total_return ctr1, store, customer
WHERE
    ctr1.ctr_total_return >
    (SELECT AVG(ctr_total_return)*1.2
     FROM customer_total_return ctr2
     WHERE
        ctr1.ctr_store_sk = ctr2.ctr_store_sk (P3)
     ) (P4) AND
     s_store_sk = ctr1.ctr_store_sk (P5)
     s_state = 'AL' (P6)
     and ctr1.ctr_customer_sk = c_customer_sk (P7)
ORDER BY c_customer_id
FETCH FIRST 100 ROWS ONLY;
```

(b) GEX weights and scores for $\mathbf{Q}_1$ predicates

	Baseline		Increased	
Predicate	Weight	Score	Weight	Score
P1	532.9	0.54	170.3	0.71
P2	534.2	0.54	170.8	0.71
P3	-2.5	-0.00	1.2	0.00
P4	3.4	0.00	0.4	0.00
P5	369.2	0.37	6.1	0.03
P6	358.6	0.36	6.0	0.03
P7	371.2	0.38	5.8	0.02

Figure 3.6: $\mathbf{Q}_1$ predicate `GEX` scores before and after increasing resource knobs

ing sensitive predicates; e.g., `s_state = 'AL'` decreases from 0.36 to 0.03, `d_year = 2002` increases from 0.54 to 0.71. The saliency scores of the `sr_returned_date_sk = d_date_sk` and `d_year = 2002` predicates shift proportionally upwards while the others decrease. This informs experts that further tuning efforts should focus on these remaining sensitive predicates.

Query Rewrite. Query rewrites transform SQL queries into equivalent forms that execute more efficiently [8]. Effective rewrites can simplify query structure, enable better plan selection, and improve performance. In practice, query rewrites can be implemented manually by experts, or automatically using registry variables that trigger predefined transformations [9]. These registry variables, often treated as configuration "knobs," can be tuned by automatic systems [10], [11], [13] to recommend optimal

rewrites for specific workloads.

Manual query rewrites require deep expertise and a detailed understanding of both the query and its execution plan. Experts need to carefully examine query execution plans to identify inefficiencies, such as costly joins or redundant operations. This process can be labor-intensive, especially for complex workloads with numerous queries. `GEX` simplifies this process by using saliency maps to identify the areas of a query with the greatest impact on optimizer cost estimates. These maps guide experts by highlighting critical query components, such as predicates in the SQL or operators within the QEP, that are prime candidates for rewrites. They can then manually rewrite those portions or match their pattern to existing defined rewrites, which can be toggled using registry variables.

In automated tuning scenarios, `GEX` complements knob-tuning systems like `Db2une` [10], which can treat registry variables as tunable parameters. Such systems may recommend query rewrites based on cost estimation patterns observed across a workload. `GEX` serves as a validation layer, allowing experts to assess the effectiveness of these recommendations by correlating them with saliency insights. This ensures that automated rewrite suggestions are both actionable and aligned with workload characteristics.

To illustrate the value of `GEX` for guiding query rewrites, consider TPC-DS $\mathbf{Q}_{67}$, Appendix 6.2, which calculates the top performing products by sales in each product category over twelve months, broken down by store and product attributes. By default, the QEP for $\mathbf{Q}_{67}$ employs a `ROLLUP` operator to compute aggregations at multiple attribute levels. In the original QEP, as shown in Fig. 3.7 (left), `GEX`'s saliency maps highlights a specific hash-join with a high saliency score, signaling a bottle-

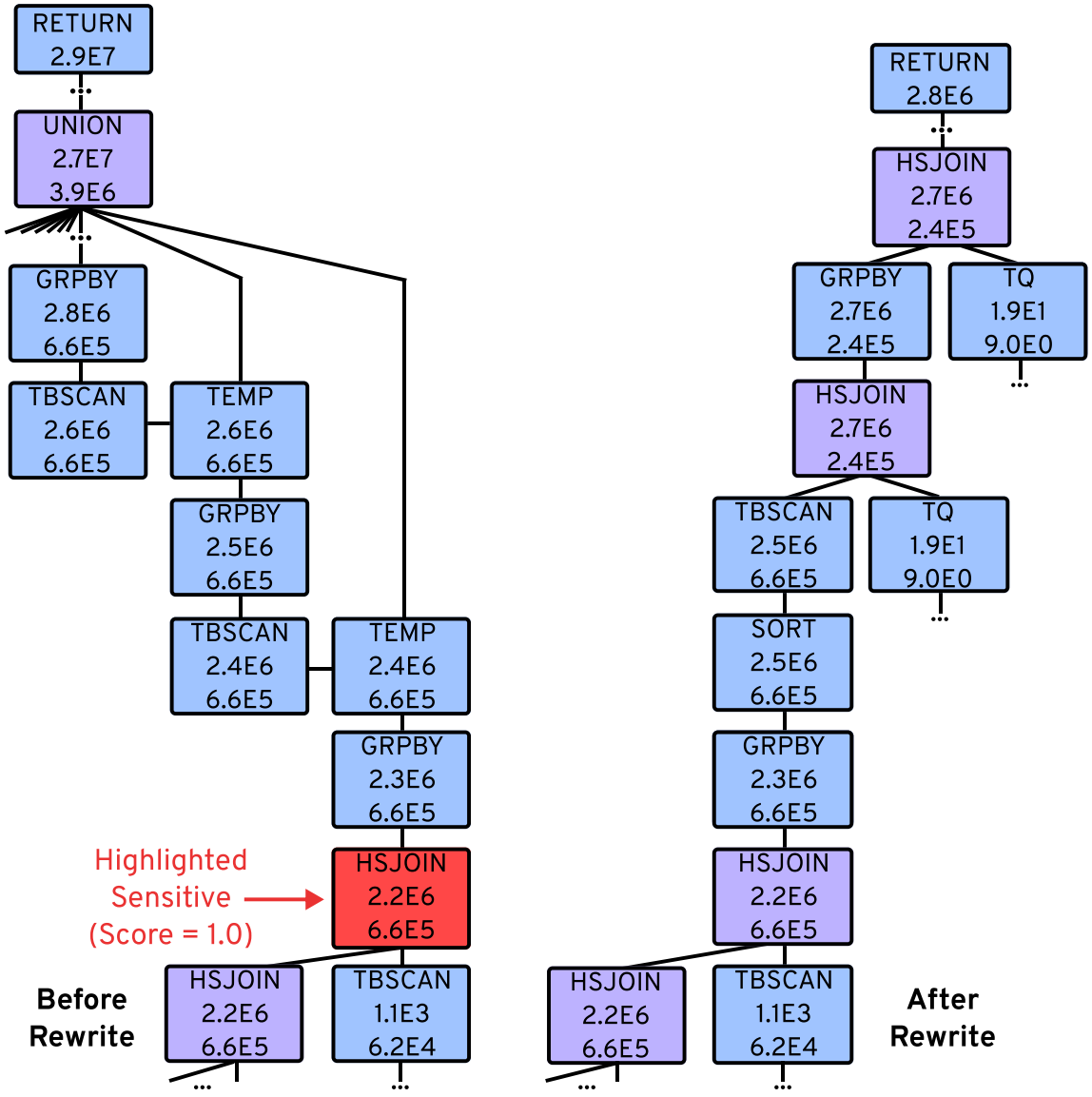

Figure 3.7: TPC-DS $\mathbf{Q}_{67}$ before and after rewrite. © 2025 IEEE

neck or a particularly sensitive operation in the plan. Each level of the `ROLLUP` was processed sequentially, with intermediate results stored in temporary tables. These intermediate results of this highlighted hash-join repeatedly have group-by, table scans, and temp operations applied, then are combined using a 9-way `UNION` operation. This incurred significant I/O overhead, as the data are repeatedly processed, written to, and read from disk. Guided by `GEX`, experts rewrote the query using registry variables to optimize the handling of the `ROLLUP` operator. The rewritten query replicates the input data across parallel streams, allowing all levels of the `ROLLUP` to be processed independently and simultaneously. This resulted in a more efficient plan, as seen on the right of Fig. 3.7, reducing estimated costs by a factor of 10 and execution time from 106.4 to 56.2 seconds (Section 4.3).

Chapter 4

Experimental Evaluation

We evaluate GEX on using a 100GB scale TPC-DS benchmark workload to demonstrate its ability to provide interpretable insights and improve workload performance. The experiments were executed on a machine with an Intel i7-8565U CPU and 32GB of RAM. All GEX scores were generated with $n = 1000$ samples with a sampling variance of $0.5*$ the original filter factor. GEX saliency score generation time for a 10 query subset of TPC-DS used in our experiments was 23 minutes 13.2 seconds. Sampling accounted for nearly all runtime, linear model training time, and saliency map generation were negligible.

4.1 Manual Tuning with Statistical Views

To evaluate GEX's ability to assist in manual statistical view selection and demonstrate their relevance to subsequent knob tuning, two experts were tasked with creating five statistical views for a randomly selected subset of ten queries from the TPC-DS benchmark. The first expert, referred to as the *Db2 & Workload Expert*, is a

Table 4.1: Effect of statistical views on runtimes (in secs). © 2025 IEEE

		Db2 & Workload Expert		Saliency-Inf. Expert	
Query	No Views	Initial	Re-Tuned	Initial	Re-Tuned
$\mathbf{Q}_{1}$	2.4	2.0	1.9	1.8	2.3
$\mathbf{Q}_{4}$	260.4	110.4	92.5	98.1	87.4
$\mathbf{Q}_{5}$	27.3	28.8	26.4	28.4	26.4
$\mathbf{Q}_{11}$	139.4	50.3	43.2	37.1	39.9
$\mathbf{Q}_{23}$	255.5	242.5	223.1	200.7	209.2
$\mathbf{Q}_{38}$	58.9	24.8	25.0	32.8	25.8
$\mathbf{Q}_{51}$	86.8	79.0	52.9	75.7	47.0
$\mathbf{Q}_{64}$	51.3	42.9	42.2	45.4	44.2
$\mathbf{Q}_{74}$	67.9	26.7	19.1	21.5	21.1
$\mathbf{Q}_{97}$	54.9	20.3	15.8	19.3	19.6
total	**1004.8**	**627.7**	**542.1**	**560.8**	**522.9**

senior IBM expert with more than 20 years of experience specializing in the `IBM Db2` optimizer and extensive knowledge of TPC-DS benchmarking, who relied on manual analysis to define statistical views. This is a time-consuming process due to the detailed exploration required to identify relevant columns and predicates. The second expert, referred to as the *Saliency-Informed Expert*, is a junior professional. Their goal was to define statistical views on the most cardinality-sensitive joins of the most expensive queries of the workload. Their approach was to order queries by run time performance then use `GEX`'s saliency maps to identify joins that contributed the columns of high scoring predicates, then define views for those joins. Knob tuning was applied manually to the sort heap and buffer pool knobs using a simple grid search for the lowest cost estimate at lowest memory allocation, with manual correction in case of regressions. Table 4.1 presents the query runtimes under 5 configurations: baseline with *no statistical views*; using *Db2 & Workload Expert* and using *Saliency-Informed Expert* with the *initial* knob tuning configurations; and using *re-tuned* configurations with knob tuning for each expert. As shown in the table, for

the initial knob tuning configurations, *Saliency-Informed Expert* achieved a 44.1% runtime reduction compared to baseline, outperforming the *Db2 & Workload Expert*, which achieved a 37.5% reduction. This underscores `GEX`'s ability to reduce manual effort while delivering high-quality optimizations.

When re-tuned after applying views, *Saliency-Informed Expert* achieved a further reduction to 48%, and *Db2 & Workload Expert* improved to 46%. For several queries, most notably $\mathbf{Q}_4$ and $\mathbf{Q}_{51}$, this re-tuning resulted in significant improvements in runtime. These results demonstrate the efficacy of `GEX` for informing statistical views in combination with manual knob tuning.

To evaluate our system's ability to inform stat view selection, we compare the recommendations of two DBAs. A "TPC-DS expert" with years of experience working with the TPC-DS benchmark and a "saliency-informed expert" without extensive TPC-DS experience, and is instead informed by our saliency maps. Both experts were instructed to define 5 stat views for a random 10-query subset of TPC-DS. The saliency-informed expert ordered the queries by runtime and then identified the most salient predicates in each query. Views were defined to cover the resulting columns of the most salient join and select predicates. The resulting 5 views are our "Saliency Views". The TPC-DS expert was tasked with defining the best 5 stat views based on their familiarity with the workload to create "Expert Views".

Table 4.1 reports the runtime in seconds for each query in the 10-query test set. For this experiment, we tune the knobs with a simple heuristic. We select the settings that return the lowest cost estimate at the lowest memory allocation from a grid search of knob values. The first column shows the TPC-DS query number. **No Views** reports the runtime for each query without views at "default" buffer pool and

sort heap settings. **Expert** and **Saliency** record the results after applying stat views from the respective DBAs without updating knob settings. The total run times for these columns show 37.5% and 44.1% decreases respectively, showing the saliency-informed expert achieved greater performance despite their unfamiliarity with the workload. Improvements are attributed to improved optimizer plan selection due to more accurate cardinality estimation. **Expert Re-Tuned** and **Saliency Re-Tuned** show the results after applying views then re-tuning the knobs. For several queries, most notably $\mathbf{Q}_4$ and $\mathbf{Q}_{51}$, this re-tuning resulted in improvements in runtime. However, for queries $\mathbf{Q}_{38}$ and $\mathbf{Q}_{64}$ re-tuning caused regressions (shown in parenthesis) with both expert and saliency views.

Regressions were due to a reduction in knob settings after the optimizer yielded a slightly lower cost estimate at lower settings. We correct this by returning the knob settings to the previous value. This results in an overall decrease in runtime greater than only applying stat views, 46%, and 48% for expert and saliency views respectively. Uncorrected total run times are shown in parenthesis. These results show the potential improvement gains from knobs tuning after applying stat views. However, they also highlight the risk of regressions and the value of having a DBA in-the-loop to identify and correct them. To further evaluate the effect of stat views on knob-tuning we examine their effects on an automatic knob-tuning system in the following section.

4.1.1 Automatic Statistical Views

To evaluate the ability of `GEX` to capture useful information for the generation of statistical views independent of the knob tuning task and expert interpretation. We

Table 4.2: Query run times for automatic & expert stat views in seconds

Views	Q1	Q4	Q5	Q11	Q23	Q38	Q51	Q64	Q74	Q97	**total**
None	1.4	189.8	31.2	139.3	216.9	45.0	50.7	40.3	46.7	20.9	782.2
Expert	1.3	82.3	20.0	31.4	198.3	29.8	44.5	38.9	18.1	17.2	481.8
Auto	1.3	79.8	18.4	31.1	187.9	22.5	43.5	37.6	19	20.8	461.9

compare the previous *Saliency-Informed Expert* statistical views with a top five set of views automatically generated by `GEX`. For the same set of 10 queries, we compare run times with identical knob settings, with both buffer pool and sort heap fixed to 100,000 pages. These large resource allocations were selected to highlight the effect of the defined statistical views on plan selection and reduce the effect of resource constraints on run time. Table 4.2 shows the results in seconds for each query and the total workload after applying the sets of statistical views. The table shows that both methods of view recommendation result in significant improvements over the configuration with no views. The automatic `GEX` views slightly outperform the expert-defined views at the per-query and workload level, further validating our claim that our system captures and highlights actionable information.

4.2 Automated Tuning with Knobs

`GEX`'s query saliency maps were evaluated in the context of automatic knob tuning using `Db2une`. To assess this, an initial `Db2une` model was trained using baseline cost estimates. `GEX` was then applied to generate saliency maps for the TPC-DS queries, highlighting predicates most sensitive to cost inaccuracies. These insights were used to guide the creation of five statistical views targeting highly-sensitive areas. `Db2une` was retrained with the updated cost estimates derived from these statistical views, and

the knob recommendations were re-evaluated. This process was repeated iteratively for five, ten, and fifteen statistical views, re-training `Db2une` after each iteration. To validate the recommendations of the automatic tuning system, the queries were re-evaluated using `GEX` after applying the knob settings.

Table 4.3: Targeted stat views for automatic knobs tuning. © 2025 IEEE

# Stat views	Baseline Model	Retrained Models	Reduction From Initial
0 (Initial)	3434s	/	/
5	2904s	2146s	37%
10	2551s	2001s	41%
15	2346s	1967s	42%

The results are presented in Table 4.3, comparing execution times under the knob recommendations of the *Baseline Model* (trained without statistical views) and recommendations of the *Retrained Models* (incorporating saliency-informed statistical views). The addition of the first five statistical views led to a 37% reduction in execution time. Further iterations with ten and fifteen statistical views resulted in additional reductions, achieving 41% and 42% decreases in runtime, respectively.

The improved statistics from views corrected underestimations in cardinality, particularly for predicates involving large fact tables, such as `store_sales` and `catalog_sales`, joined with `date_dim`. This adjustment led to more accurate memory allocation recommendations by `Db2une`, such as increasing buffer-pool and sort-heap sizes. The diminishing returns observed with increasing the number of views reflect the decreasing utility of each additional view towards correcting cost inaccuracies.

This evaluation demonstrates that a small number of well-targeted statistical views, guided by `GEX`'s saliency maps, can significantly enhance the effectiveness of knob tuning. The approach bridges the gap between manual efforts and automated

tuning, ensuring that Db2une's recommendations align closely with the specific characteristics of the workload.

4.3 Informed Query Rewrites

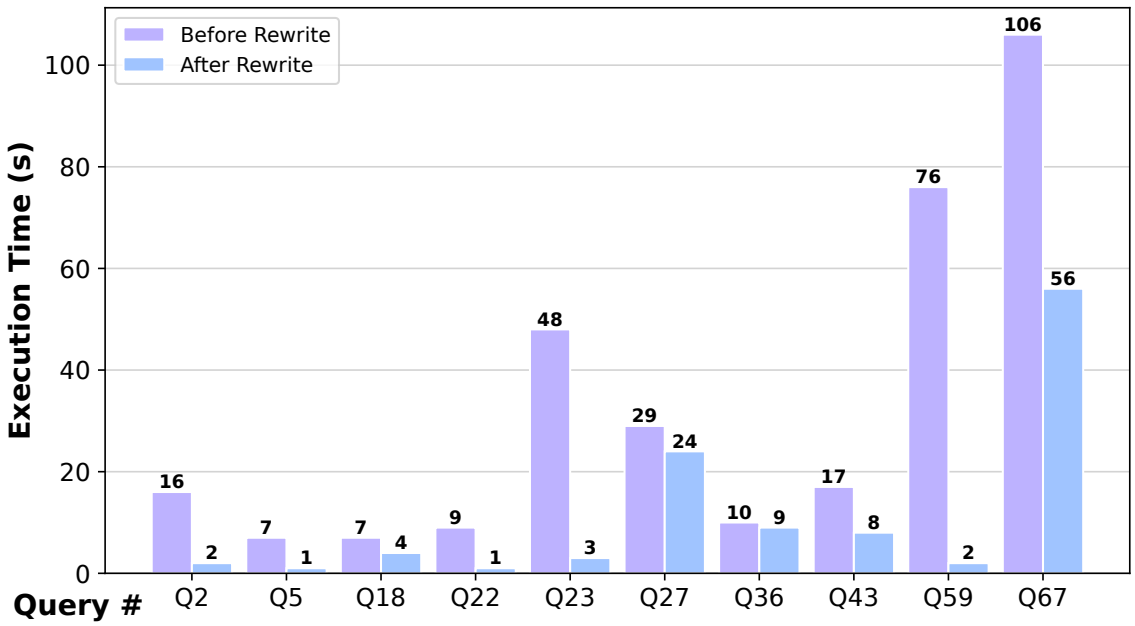

Figure 4.1: TPC-DS queries before and after informed rewrites. © 2025 IEEE

We demonstrate how GEX guides experts in query rewrites by ranking queries from the entire TPC-DS workload based on their GEX-assigned saliency scores. This ranking allowed experts to quickly identify strong candidates for rewrites by applying the appropriate registry variables. To determine effective rewrites, experts analyzed highlighted predicates in the query saliency maps and corresponding operators in the QEP saliency maps. As shown in Fig. 4.1, these guided rewrites dramatically reduced the total execution time for ten queries, resulting in an overall 66% decrease in runtime.

GEX identified predicates in SQL queries linked to expensive ROLLUP operations and operators in plans preceding the chain of sequential GROUP BY operations caused by that ROLLUP. This issue was addressed by applying a rewrite to enable parallel stream processing of the ROLLUP levels, reducing runtime. This approach was applied to $\mathbf{Q}_5$, $\mathbf{Q}_{18}$, $\mathbf{Q}_{22}$, $\mathbf{Q}_{36}$, $\mathbf{Q}_{67}$. Additional rewrites included early aggregation for $\mathbf{Q}_2$, $\mathbf{Q}_{43}$, and $\mathbf{Q}_{59}$, as well as enforcing DISTINCT on the results of IN subqueries for $\mathbf{Q}_{23}$. This evaluation highlights GEX's ability to integrate transparency into query rewrites.

Chapter 5

Related Work

Significant work has been developed in the field of explainable ai for providing explanations for black-box models. As mentioned throughout this work, saliency methods such as SHAP [19] and LIME [18] use surrogate modeling to generate saliency scores. These approaches use sampling techniques which are inapplicable to our domain of SQL queries. SHAP samples by replacing features with random features from other points in the dataset while for continuous features LIME samples continuous feature values from a distribution which it computes from the training dataset. In both cases, we lack a dataset with consistent features, e.g. many queries with the same predicates and varying filter factors. In addition to sampling-based saliency approaches, other well-known approaches [28] such as ϵ-LRP [38], deepLIFT [29] and integrated gradients [39], and use the gradients of the systems they explain to propagate feature saliency. We are unable to benefit from these techniques due to the lack of gradients provided by the classical cost-based optimizer.

Counterfactual explanations have also been explored as a method of explaining black boxes [27], [40], [41]. These explanations provide insight into a model's decision-

making by providing the minimal change needed to be made to the input of the model to change an outcome. This type of explanation has been applied in the data systems domain to explain ML-based systems such as SQL injection detection [42]. Separately from their application to AI systems, counterfactual explanations have been adapted to explore causal relationships within data systems through SQL queries [41], [43].

In addition to saliency and counterfactual explanations that provide explanations with respect to a specific sample input data, *rule-based* explanations provide a set of rules which characterize model behaviour in a human-interpretable manner. These techniques can vary greatly in their rule generation techniques and presentation. For example: surrogate modeling using decision trees is used to provide inherently interpretable rule-based representations [44], explanation tables use an information gain metric to identify minimal rules that have the greatest data coverage to support model decisions [45], and existing counterfactual explanations have been used to define rules about model behaviour [46]. We discuss extending our system with counterfactual and rule-based explanations for a cost-based optimizer in future work 6.2.

Chapter 6

Conclusions

6.1 Summary

We introduced GEX, a system that uses novel extensions to XAI techniques to provide interpretable insights into cost-based database optimizers. This work serves as a successful initial exploration into using perturbations of cardinality estimates at key points of a query to gain meaningful insight into query properties. In addition, it introduces to our knowledge the first use of sampling via normal distributions with a variance as a percentage of the original value for continuously valued features. We find that this sampling approach, along with the training of a surrogate model, provided useful saliency scores.

By generating saliency maps with surrogate models, GEX highlights key query components to guide experts in downstream tuning tasks. The novel metrics for cost estimate sensitivity with respect to changes in cardinality are particularly relevant for the task of defining statistical views for a workload. Shown in our experimental results, GEX can be used to guide statistical view selection that exceeds domain expert

results. IBM is strongly interested in systems that support experts in optimizing complex workloads. Due to this, IBM has expressed interest in this work, both as a standalone tool and as a source of information for automatic tools for both statistical view recommendation, query optimization, and other system configuration tasks.

6.2 Future Work

`GEX` is designed to create local linear surrogate models. However, the use of linear models constrains the expressiveness of our surrogate model, limiting its ability to represent complex relationships. It could be beneficial to the faithfulness of our model's representation to be able to better fit a wider range of optimizer behaviour. Therefore, we intend to explore extensions to the surrogate modeling, including the addition of interaction and higher-order terms. This will necessitate research into how the weights for these additional terms can be attributed to the saliency score of features. In addition to this, we aim to further develop our approach for sampling data for training local surrogate models, refining the currently manually tuned variance of the normal distribution sampling.

`GEX` is currently focused solely on generating saliency-based explanations. We plan on expanding the scope of the project to incorporate "what-if" analyses [40] to identify decision boundaries within the structure of query plans. Counterfactual explanations for changes in plan structure could be generated through perturbations to query cardinality and knob settings. These would provide information about the minimal change needed to query cardinality or system settings to produce a different query execution plan. Incorporating perturbations of system knobs would allow experts to identify the minimum change to system settings required to change the execution

of a query. Perturbations in query cardinality could be used to show tolerance to estimation error or data growth for plan generation. This work may be extended with rule-based explanations by mining patterns in relationships between `GEX` scores, query operators, and associated changes in system performance when highlighted query operations are addressed with corresponding settings. These rules could capture commonly occurring sensitive patterns within query plans and their tuning solutions, providing guidelines for system optimization.

Bibliography

[1] V. Leis, A. Gubichev, A. Mirchev, P. A. Boncz, A. Kemper, and T. Neumann, "How good are query optimizers, really?" *Proc. VLDB Endow.*, vol. 9, no. 3, pp. 204–215, 2015. DOI: 10.14778/2850583.2850594. [Online]. Available: http://www.vldb.org/pvldb/vol9/p204-leis.pdf.

[2] W. Tan, M. Alhamid, M. Kalil, *et al.*, "Query predicate selectivity using machine learning in db2®," in *Proceedings of the 31st Annual International Conference on Computer Science and Software Engineering*, ser. CASCON '21, ACM, 2021, pp. 143–152. DOI: 10.5555/3507788.3507808. [Online]. Available: https://dl.acm.org/doi/10.5555/3507788.3507808.

[3] Y. Han, H. Wang, L. Chen, *et al.*, "Bytecard: Enhancing bytedance's data warehouse with learned cardinality estimation," in *Companion of the 2024 International Conference on Management of Data*, ser. SIGMOD '24, ACM, 2024, pp. 41–54. DOI: 10.1145/3626246.3653376. [Online]. Available: https://doi.org/10.1145/3626246.3653376.

[4] 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.

[5] IBM Db2 Documentation, *Statistical views*, Accessed: 2024-11-20, 2024. [Online]. Available: https://www.ibm.com/docs/en/db2/11.1?topic=optimization-statistical-views.

[6] IBM Db2 Documentation, *Configuration parameters summary*, Accessed: 2024-11-20, 2024. [Online]. Available: https://www.ibm.com/docs/en/db2/12.1?topic=parameters-configuration-summary.

[7] X. Zhao, X. Zhou, and G. Li, "Automatic database knob tuning: A survey," *IEEE Trans. Knowl. Data Eng.*, vol. 35, no. 12, pp. 12 470–12 490, 2023. DOI: 10.1109/TKDE.2023.3266893. [Online]. Available: https://doi.org/10.1109/TKDE.2023.3266893.

[8] G. Damasio, V. Corvinelli, P. Godfrey, *et al.*, "Guided automated learning for query workload re-optimization," *PVLDB*, vol. 12, no. 12, pp. 2010–2021, 2019.

[9] IBM Db2 Documentation, *Db2 registry and environment variables*, Accessed: 2024-11-20, 2024. [Online]. Available: https://www.ibm.com/docs/en/db2/12.1?topic=variables-registry-environment.

[10] 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.

[11] 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.

[12] W. Zhang, W. S. Lim, M. Butrovich, and A. Pavlo, "The holon approach for simultaneously tuning multiple components in a self-driving database management system with machine learning via synthesized proto-actions," *Proc. VLDB Endow.*, vol. 17, no. 11, pp. 3373–3387, 2024. DOI: 10.14778/3681954.3682007. [Online]. Available: https://doi.org/10.14778/3681954.3682007.

[13] R. Marcus, P. Negi, H. Mao, N. Tatbul, M. Alizadeh, and T. Kraska, "Bao: Making learned query optimization practical," in *Proceedings of the 2021 International Conference on Management of Data*, ser. SIGMOD '21, ACM, 2021, pp. 1275–1288. DOI: 10.1145/3448016.3452838. [Online]. Available: https://doi.org/10.1145/3448016.3452838.

[14] 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, 2022. DOI: 10.14778/3565816.3565824. [Online]. Available: https://www.vldb.org/pvldb/vol16/p216-perera.pdf.

[15] T. Yu, Z. Zou, W. Sun, and Y. Yan, "Refactoring index tuning process with benefit estimation," *Proc. VLDB Endow.*, vol. 17, no. 7, pp. 1528–1541, 2024. [Online]. Available: https://www.vldb.org/pvldb/vol17/p1528-zou.pdf.

[16] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *CoRR*, vol. abs/1707.06347, 2017. arXiv: 1707.06347. [Online]. Available: http://arxiv.org/abs/1707.06347.

[17] T. P. Lillicrap, J. J. Hunt, A. Pritzel, *et al.*, "Continuous control with deep reinforcement learning," in *4th International Conference on Learning Representations, ICLR 2016*, 2016. [Online]. Available: http://arxiv.org/abs/1509.02971.

[18] M. T. Ribeiro, S. Singh, and C. Guestrin, ""why should I trust you?": Explaining the predictions of any classifier," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ACM, 2016, pp. 1135–1144. DOI: 10.1145/2939672.2939778. [Online]. Available: https://doi.org/10.1145/2939672.2939778.

[19] S. M. Lundberg and S. Lee, "A unified approach to interpreting model predictions," in *Advances in Neural Information Processing Systems*, 2017, pp. 4765–4774. [Online]. Available: https://arxiv.org/abs/1705.07874.

[20] M. Ali, A. M. Khattak, Z. Ali, *et al.*, "Estimation and interpretation of machine learning models with customized surrogate model," *Electronics*, vol. 10, no. 23, 2021, ISSN: 2079-9292. DOI: 10.3390/electronics10233045. [Online]. Available: https://www.mdpi.com/2079-9292/10/23/3045.

[21] M. Bianchi, A. De Santis, A. Tocchetti, and M. Brambilla, "Interpretable network visualizations: A human-in-the-loop approach for post-hoc explainability of cnn-based image classification," in *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence*, ser. IJCAI-2024, International

Joint Conferences on Artificial Intelligence Organization, Aug. 2024, pp. 3715–3723. DOI: 10.24963/ijcai.2024/411. [Online]. Available: http://dx.doi.org/10.24963/ijcai.2024/411.

[22] K. Simonyan, A. Vedaldi, and A. Zisserman, "Deep inside convolutional networks: Visualising image classification models and saliency maps," in *2nd International Conference on Learning Representations, ICLR 2014*, 2014. [Online]. Available: http://arxiv.org/abs/1312.6034.

[23] V. Borisov, T. Leemann, K. Seßler, J. Haug, M. Pawelczyk, and G. Kasneci, "Deep neural networks and tabular data: A survey," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 35, no. 6, pp. 7499–7519, 2024. DOI: 10.1109/TNNLS.2022.3229161. [Online]. Available: https://doi.org/10.1109/TNNLS.2022.3229161.

[24] A. Chai, A. Vezvaei, L. Golab, *et al.*, "EAGER: explainable question answering using knowledge graphs," in *Proceedings of the 6th Joint Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*, ACM, 2023, pp. 1–5. DOI: 10.1145/3594778.3594877. [Online]. Available: https://doi.org/10.1145/3594778.3594877.

[25] A. Chai, A. Bianchi, V. Corvinelli, P. Godfrey, J. Szlichta, and C. Zuzarte, "Gex: Guiding expert tuning with explainable ai," in *Proceedings of the 41st IEEE International Conference on Data Engineering*, ser. ICDE '25, Hong Kong SAR, China: Institute of Electrical and Electronics Engineers, 2025, pp. 1–8.

[26] M. Poess, B. Smith, L. Kollar, and P. Larson, "Tpc-ds, taking decision support benchmarking to the next level," in *Proceedings of the 2002 ACM SIGMOD*

International Conference on Management of Data, ser. SIGMOD '02, Madison, Wisconsin: Association for Computing Machinery, 2002, pp. 582–587, ISBN: 1581134975. DOI: 10.1145/564691.564759. [Online]. Available: https://doi.org/10.1145/564691.564759.

[27] C. Molnar, *Interpretable Machine Learning, A Guide for Making Black Box Models Explainable*, 2nd ed. BOOKDOWN, 2022. [Online]. Available: https://christophm.github.io/interpretable-ml-book.

[28] M. Ancona, E. Ceolini, C. Öztireli, and M. Gross, *Towards better understanding of gradient-based attribution methods for deep neural networks*, 2018. arXiv: 1711.06104 [cs.LG]. [Online]. Available: https://arxiv.org/abs/1711.06104.

[29] A. Shrikumar, P. Greenside, and A. Kundaje, *Learning important features through propagating activation differences*, 2019. arXiv: 1704.02685 [cs.CV]. [Online]. Available: https://arxiv.org/abs/1704.02685.

[30] L. Shapley, "7. a value for n-person games. contributions to the theory of games ii (1953) 307-317.," in *Classics in Game Theory*, H. W. Kuhn, Ed. Princeton: Princeton University Press, 1997, pp. 69–79, ISBN: 9781400829156. DOI: doi:10.1515/9781400829156-012. [Online]. Available: https://doi.org/10.1515/9781400829156-012.

[31] I. D. Documentation, *Configuration parameters summary*, https://www.ibm.com/docs/en/db2/9.7?topic=parameters-sortheap-sort-heap-size, 2021.

[32] A. Albrecht, *Sqlparse (version 0.5.2)*, Accessed: 2024-11-25, 2024. [Online]. Available: https://pypi.org/project/sqlparse/.

[33] IBM Db2 Documentation, *Search conditions*, Accessed: 2024-11-20, 2024. [Online]. Available: https://www.ibm.com/docs/en/db2/12.1?topic=predicates-search-conditions#sdx-synid_selectivity.

[34] C. A. Galindo-Legaria, M. Joshi, F. Waas, and M. Wu, "Statistics on views," in *Proceedings of 29th International Conference on Very Large Data Bases, VLDB 2003*, 2003, pp. 952–962. DOI: 10.1016/B978-012722442-8/50089-6. [Online]. Available: http://www.vldb.org/conf/2003/papers/S28P03.pdf.

[35] B. Ding, S. Das, W. Wu, S. Chaudhuri, and V. R. Narasayya, "Plan stitch: Harnessing the best of many plans," *Proc. VLDB Endow.*, vol. 11, no. 10, pp. 1123–1136, 2018. DOI: 10.14778/3231751.3231761. [Online]. Available: http://www.vldb.org/pvldb/vol11/p1123-ding.pdf.

[36] 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, VLDB 2004*, 2004, pp. 1087–1097. DOI: 10.1016/B978-012088469-8.50095-4. [Online]. Available: http://www.vldb.org/conf/2004/IND4P1.PDF.

[37] H. Yuan, G. Li, L. Feng, J. Sun, and Y. Han, "Automatic view generation with deep learning and reinforcement learning," in *36th IEEE International Conference on Data Engineering, ICDE 2020*, IEEE, 2020, pp. 1501–1512. DOI: 10.1109/ICDE48307.2020.00133. [Online]. Available: https://doi.org/10.1109/ICDE48307.2020.00133.

[38] S. Bach, A. Binder, G. Montavon, F. Klauschen, K.-R. Müller, and W. Samek, "On pixel-wise explanations for non-linear classifier decisions by layer-wise relevance propagation," *PloS one*, vol. 10, no. 7, e0130140, 2015.

[39] M. Sundararajan, A. Taly, and Q. Yan, "Axiomatic attribution for deep networks," in *International conference on machine learning*, PMLR, 2017, pp. 3319–3328.

[40] S. Dandl, C. Molnar, M. Binder, and B. Bischl, "Multi-objective counterfactual explanations," in *Parallel Problem Solving from Nature - PPSN XVI*, ser. Lecture Notes in Computer Science, vol. 12269, Springer, 2020, pp. 448–469. DOI: 10.1007/978-3-030-58112-1_31. [Online]. Available: https://doi.org/10.1007/978-3-030-58112-1%5C_31.

[41] P. M. VanNostrand, H. Zhang, D. M. Hofmann, and E. A. Rundensteiner, "Facet: Robust counterfactual explanation analytics," *Proc. ACM Manag. Data*, vol. 1, no. 4, Dec. 2023. DOI: 10.1145/3626729. [Online]. Available: https://doi.org/10.1145/3626729.

[42] S. Roy and D. Suciu, "A formal approach to finding explanations for database queries," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '14, Snowbird, Utah, USA: Association for Computing Machinery, 2014, pp. 1579–1590, ISBN: 9781450323765. DOI: 10.1145/2588555.2588578. [Online]. Available: https://doi.org/10.1145/2588555.2588578.

[43] A. Meliou, S. Roy, and D. Suciu, "Causality and explanations in databases," *Proc. VLDB Endow.*, vol. 7, no. 13, pp. 1715–1716, Aug. 2014, ISSN: 2150-8097.

DOI: 10.14778/2733004.2733070. [Online]. Available: https://doi.org/10.14778/2733004.2733070.

[44] F. Di Castro and E. Bertini, "Surrogate decision tree visualization.," in *IUI Workshops*, 2019.

[45] K. El Gebaly, G. Feng, L. Golab, F. Korn, and D. Srivastava, "Explanation tables," *Sat*, vol. 5, p. 14, 2018.

[46] Z. Geng, M. Schleich, and D. Suciu, *Computing rule-based explanations by leveraging counterfactuals*, 2022. arXiv: 2210.17071 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2210.17071.

Copyright

Appendix

TPC-DS [26] Query 1

```
with customer_total_return as
  (select
     sr_customer_sk as ctr_customer_sk, sr_store_sk as ctr_store_sk, sum(SR_REFUNDED_CASH)
     as ctr_total_return
   from
     store_returns, date_dim
   where
     sr_returned_date_sk = d_date_sk and d_year =2002
   group by
     sr_customer_sk, sr_store_sk
  )
select
  c_customer_id
from
  customer_total_return ctr1, store, customer
where
  ctr1.ctr_total_return >
  (select
     avg(ctr_total_return)*1.2
   from
     customer_total_return ctr2
   where
     ctr1.ctr_store_sk = ctr2.ctr_store_sk
  ) and
  s_store_sk = ctr1.ctr_store_sk and s_state = 'AL' and  ctr1.ctr_customer_sk = c_customer_sk
```

```
order by
  c_customer_id
fetch first 100 rows only;
```

TPC-DS Query 2

```
with wscs as
  (select
     sold_date_sk, sales_price
   from
     (select
        ws_sold_date_sk sold_date_sk, ws_ext_sales_price sales_price
      from
        web_sales
      union all
      select
        cs_sold_date_sk sold_date_sk, cs_ext_sales_price sales_price
      from
        catalog_sales
     )
  ),
  wswscs as
  (select
     d_week_seq,
     sum(
         case
         when (d_day_name='Sunday')
         then sales_price
         else null end) sun_sales,
     sum(
         case
         when (d_day_name='Monday')
         then sales_price
         else null end) mon_sales,
     sum(
         case
```

```
        when (d_day_name='Tuesday')
        then sales_price
        else null end) tue_sales,
    sum(
        case
        when (d_day_name='Wednesday')
        then sales_price
        else null end) wed_sales,
    sum(
        case
        when (d_day_name='Thursday')
        then sales_price
        else null end) thu_sales,
    sum(
        case
        when (d_day_name='Friday')
        then sales_price
        else null end) fri_sales,
    sum(
        case
        when (d_day_name='Saturday')
        then sales_price
        else null end) sat_sales
  from
    wscs, date_dim
  where
    d_date_sk = sold_date_sk
  group by
    d_week_seq
 )
select
 d_week_seq1, round(sun_sales1/sun_sales2, 2), round(mon_sales1/mon_sales2, 2),
 round(tue_sales1/tue_sales2, 2),  round(wed_sales1/wed_sales2, 2),
 round(thu_sales1/thu_sales2, 2), round(fri_sales1/fri_sales2, 2),
 round(sat_sales1/sat_sales2, 2)
from
 (select
```

```
    wswscs.d_week_seq d_week_seq1, sun_sales sun_sales1, mon_sales mon_sales1, tue_sales tue_sales1,
    wed_sales wed_sales1, thu_sales thu_sales1, fri_sales fri_sales1, sat_sales sat_sales1
   from
    wswscs, date_dim
   where
    date_dim.d_week_seq = wswscs.d_week_seq and d_year = 2000
  ) y,
  (select
    wswscs.d_week_seq d_week_seq2, sun_sales sun_sales2, mon_sales mon_sales2,
    tue_sales tue_sales2, wed_sales wed_sales2, thu_sales thu_sales2,
    fri_sales fri_sales2, sat_sales sat_sales2
   from
    wswscs, date_dim
   where
    date_dim.d_week_seq = wswscs.d_week_seq and d_year = 2000+1
  ) z
where
  d_week_seq1=d_week_seq2-53
order by
  d_week_seq1;
```

TPC-DS Query 4

```
with year_total as
  (select
    c_customer_id customer_id, c_first_name customer_first_name, c_last_name customer_last_name,
    c_preferred_cust_flag customer_preferred_cust_flag, c_birth_country customer_birth_country,
    c_login customer_login, c_email_address customer_email_address, d_year dyear,
    sum(((ss_ext_list_price-ss_ext_wholesale_cost-ss_ext_discount_amt)+ss_ext_sales_price)/2) year_total,
    's' sale_type
   from
    customer, store_sales, date_dim
   where
    c_customer_sk = ss_customer_sk and  ss_sold_date_sk = d_date_sk
   group by
    c_customer_id, c_first_name, c_last_name, c_preferred_cust_flag, c_birth_country, c_login,
```

```
    c_email_address, d_year
  union all
  select
    c_customer_id customer_id, c_first_name customer_first_name,  c_last_name customer_last_name,
    c_preferred_cust_flag customer_preferred_cust_flag,  c_birth_country customer_birth_country,
    c_login customer_login, c_email_address customer_email_address, d_year dyear,
    sum((((cs_ext_list_price-cs_ext_wholesale_cost-cs_ext_discount_amt)+cs_ext_sales_price)/2))
    year_total, 'c' sale_type
  from
    customer, catalog_sales, date_dim
  where
    c_customer_sk = cs_bill_customer_sk and cs_sold_date_sk = d_date_sk
  group by
    c_customer_id, c_first_name, c_last_name, c_preferred_cust_flag, c_birth_country, c_login,
    c_email_address, d_year
  union all
  select
    c_customer_id customer_id, c_first_name customer_first_name, c_last_name customer_last_name,
    c_preferred_cust_flag customer_preferred_cust_flag, c_birth_country customer_birth_country,
    c_login customer_login, c_email_address customer_email_address, d_year dyear,
    sum((((ws_ext_list_price-ws_ext_wholesale_cost-ws_ext_discount_amt)+ws_ext_sales_price)/2))
    year_total, 'w' sale_type
  from
    customer, web_sales, date_dim
  where
    c_customer_sk = ws_bill_customer_sk and ws_sold_date_sk = d_date_sk
  group by
    c_customer_id, c_first_name, c_last_name, c_preferred_cust_flag, c_birth_country, c_login,
    c_email_address, d_year
 )
select
  t_s_secyear.customer_id, t_s_secyear.customer_first_name, t_s_secyear.customer_last_name,
  t_s_secyear.customer_preferred_cust_flag
from
  year_total t_s_firstyear, year_total t_s_secyear, year_total t_c_firstyear, year_total t_c_secyear,
  year_total t_w_firstyear, year_total t_w_secyear
where
```

```
  t_s_secyear.customer_id = t_s_firstyear.customer_id and
  t_s_firstyear.customer_id = t_c_secyear.customer_id and
  t_s_firstyear.customer_id = t_c_firstyear.customer_id and
  t_s_firstyear.customer_id = t_w_firstyear.customer_id and
  t_s_firstyear.customer_id = t_w_secyear.customer_id and
  t_s_firstyear.sale_type = 's' and t_c_firstyear.sale_type = 'c' and
  t_w_firstyear.sale_type = 'w' and t_s_secyear.sale_type = 's' and
  t_c_secyear.sale_type = 'c' and t_w_secyear.sale_type = 'w' and
  t_s_firstyear.dyear = 1998 and t_s_secyear.dyear = 1998+1 and
  t_c_firstyear.dyear = 1998 and t_c_secyear.dyear = 1998+1 and
  t_w_firstyear.dyear = 1998 and t_w_secyear.dyear = 1998+1 and
  t_s_firstyear.year_total > 0 and t_c_firstyear.year_total > 0 and
  t_w_firstyear.year_total > 0 and
case
when t_c_firstyear.year_total > 0
then t_c_secyear.year_total / t_c_firstyear.year_total
else null end >
case
when t_s_firstyear.year_total > 0
then t_s_secyear.year_total / t_s_firstyear.year_total
else null end and
case
when t_c_firstyear.year_total > 0
then t_c_secyear.year_total / t_c_firstyear.year_total
else null end >
case
when t_w_firstyear.year_total > 0
then t_w_secyear.year_total / t_w_firstyear.year_total
else null end
order by
  t_s_secyear.customer_id, t_s_secyear.customer_first_name,
  t_s_secyear.customer_last_name, t_s_secyear.customer_preferred_cust_flag
fetch first 100 rows only;
```

TPC-DS Query 5

```
with ssr as
  (select
     s_store_id, sum(sales_price) as sales, sum(profit) as profit, sum(return_amt) as returns,
     sum(net_loss) as profit_loss
   from
     (select
        ss_store_sk as store_sk, ss_sold_date_sk as date_sk, ss_ext_sales_price as sales_price,
        ss_net_profit as profit, cast(0 as decimal(7, 2)) as return_amt,
        cast(0 as decimal(7, 2)) as net_loss
      from
        store_sales
      union all
      select
        sr_store_sk as store_sk, sr_returned_date_sk as date_sk,
        cast(0 as decimal(7, 2)) as sales_price, cast(0 as decimal(7, 2)) as profit,
        sr_return_amt as return_amt, sr_net_loss as net_loss
      from
        store_returns
     ) salesreturns, date_dim, store
   where
     date_sk = d_date_sk and
     d_date between cast('2001-08-22' as date) and (cast('2001-08-22' as date) + 14 days) and
     store_sk = s_store_sk
   group by
     s_store_id
  ),
  csr as
  (select
     cp_catalog_page_id, sum(sales_price) as sales, sum(profit) as profit,
     sum(return_amt) as returns, sum(net_loss) as profit_loss
   from
     (select
        cs_catalog_page_sk as page_sk,  cs_sold_date_sk as date_sk,
        cs_ext_sales_price as sales_price,  cs_net_profit as profit,
        cast(0 as decimal(7, 2)) as return_amt,  cast(0 as decimal(7, 2)) as net_loss
      from
        catalog_sales
```

```
    union all
    select
      cr_catalog_page_sk as page_sk, cr_returned_date_sk as date_sk,
      cast(0 as decimal(7, 2)) as sales_price, cast(0 as decimal(7, 2)) as profit,
      cr_return_amount as return_amt, cr_net_loss as net_loss
    from
      catalog_returns
   ) salesreturns, date_dim, catalog_page
 where
   date_sk = d_date_sk and
   d_date between cast('2001-08-22' as date) and (cast('2001-08-22' as date) + 14 days) and
   page_sk = cp_catalog_page_sk
 group by
   cp_catalog_page_id
),
wsr as
(select
   web_site_id, sum(sales_price) as sales, sum(profit) as profit,
   sum(return_amt) as returns, sum(net_loss) as profit_loss
 from
   (select
      ws_web_site_sk as wsr_web_site_sk, ws_sold_date_sk as date_sk,
      ws_ext_sales_price as sales_price, ws_net_profit as profit,
      cast(0 as decimal(7, 2)) as return_amt, cast(0 as decimal(7, 2)) as net_loss
    from
      web_sales
    union all
    select
      ws_web_site_sk as wsr_web_site_sk, wr_returned_date_sk as date_sk,
      cast(0 as decimal(7, 2)) as sales_price, cast(0 as decimal(7, 2)) as profit,
      wr_return_amt as return_amt, wr_net_loss as net_loss
    from
      web_returns
      left outer join web_sales
      on (wr_item_sk = ws_item_sk and wr_order_number = ws_order_number)
   ) salesreturns, date_dim, web_site
 where
```

```
     date_sk = d_date_sk and d_date between cast('2001-08-22' as date) and
     (cast('2001-08-22' as date) + 14 days) and wsr_web_site_sk = web_site_sk
   group by
     web_site_id
  )
select
  channel, id, sum(sales) as sales, sum(returns) as returns, sum(profit) as profit
from
  (select
     'store channel' as channel, 'store' || s_store_id as id, sales, returns,
     (profit - profit_loss) as profit
   from
     ssr
   union all
   select
     'catalog channel' as channel, 'catalog_page' || cp_catalog_page_id as id, sales,
     returns, (profit - profit_loss) as profit
   from
     csr
   union all
   select
     'web channel' as channel, 'web_site' || web_site_id as id, sales, returns,
     (profit - profit_loss) as profit
   from
     wsr
  ) x
group by
  rollup (channel, id)
order by
  channel, id
fetch first 100 rows only;
```

TPC-DS Query 11

```
with year_total as
  (select
```

```
    c_customer_id customer_id, c_first_name customer_first_name, c_last_name customer_last_name,
    c_preferred_cust_flag customer_preferred_cust_flag, c_birth_country customer_birth_country,
    c_login customer_login,  c_email_address customer_email_address, d_year dyear,
    sum(ss_ext_list_price-ss_ext_discount_amt) year_total, 's' sale_type
  from
    customer, store_sales, date_dim
  where
    c_customer_sk = ss_customer_sk and  ss_sold_date_sk = d_date_sk
  group by
    c_customer_id, c_first_name, c_last_name, c_preferred_cust_flag, c_birth_country, c_login,
    c_email_address, d_year
  union all
  select
    c_customer_id customer_id, c_first_name customer_first_name, c_last_name customer_last_name,
    c_preferred_cust_flag customer_preferred_cust_flag, c_birth_country customer_birth_country,
    c_login customer_login,  c_email_address customer_email_address,
    d_year dyear, sum(ws_ext_list_price-ws_ext_discount_amt) year_total, 'w' sale_type
  from
    customer, web_sales, date_dim
  where
    c_customer_sk = ws_bill_customer_sk and  ws_sold_date_sk = d_date_sk
  group by
    c_customer_id, c_first_name, c_last_name, c_preferred_cust_flag, c_birth_country, c_login,
    c_email_address, d_year
 )
select
  t_s_secyear.customer_id, t_s_secyear.customer_first_name,
  t_s_secyear.customer_last_name, t_s_secyear.customer_preferred_cust_flag
from
  year_total t_s_firstyear, year_total t_s_secyear,
  year_total t_w_firstyear, year_total t_w_secyear
where
  t_s_secyear.customer_id = t_s_firstyear.customer_id and
  t_s_firstyear.customer_id = t_w_secyear.customer_id and
  t_s_firstyear.customer_id = t_w_firstyear.customer_id and
  t_s_firstyear.sale_type = 's' and t_w_firstyear.sale_type = 'w' and
  t_s_secyear.sale_type = 's' and t_w_secyear.sale_type = 'w' and
```

```
  t_s_firstyear.dyear = 2001 and t_s_secyear.dyear = 2001+1 and
  t_w_firstyear.dyear = 2001 and t_w_secyear.dyear = 2001+1 and
  t_s_firstyear.year_total > 0 and t_w_firstyear.year_total > 0 and
case
when t_w_firstyear.year_total > 0
then t_w_secyear.year_total / t_w_firstyear.year_total
else 0.0 end >
case
when t_s_firstyear.year_total > 0
then t_s_secyear.year_total / t_s_firstyear.year_total
else 0.0 end
order by
  t_s_secyear.customer_id, t_s_secyear.customer_first_name,
  t_s_secyear.customer_last_name, t_s_secyear.customer_preferred_cust_flag
fetch first 100 rows only;
```

TPC-DS Query 18

```
select
  i_item_id, ca_country, ca_state, ca_county,
  avg(cast(cs_quantity as decimal(12, 2))) agg1, avg(cast(cs_list_price as decimal(12, 2))) agg2,
  avg(cast(cs_coupon_amt as decimal(12, 2))) agg3, avg(cast(cs_sales_price as decimal(12, 2))) agg4,
  avg(cast(cs_net_profit as decimal(12, 2))) agg5, avg(cast(c_birth_year as decimal(12, 2))) agg6,
  avg(cast(cd1.cd_dep_count as decimal(12, 2))) agg7
from
  catalog_sales, customer_demographics cd1, customer_demographics cd2, customer,
  customer_address, date_dim, item
where
  cs_sold_date_sk = d_date_sk and cs_item_sk = i_item_sk and
  cs_bill_cdemo_sk = cd1.cd_demo_sk and cs_bill_customer_sk = c_customer_sk and
  cd1.cd_gender = 'F' and cd1.cd_education_status = 'College' and
  c_current_cdemo_sk = cd2.cd_demo_sk and c_current_addr_sk = ca_address_sk and
  c_birth_month in (1, 12, 2, 7, 6, 9) and d_year = 2002 and
  ca_state in ('TX', 'ID', 'SD', 'CT', 'VT', 'MO', 'KS')
group by
  rollup (i_item_id, ca_country, ca_state, ca_county)
```

```
order by
  ca_country, ca_state, ca_county, i_item_id
fetch first 100 rows only;
```

TPC-DS Query 22

```
select
  i_product_name, i_brand, i_class, i_category, avg(cast(inv_quantity_on_hand as double)) qoh
from
  inventory, date_dim,item
where
  inv_date_sk=d_date_sk and inv_item_sk=i_item_sk and d_month_seq between 1193 and 1193 + 11
group by
  rollup(i_product_name, i_brand, i_class, i_category)
order by
  qoh, i_product_name, i_brand, i_class, i_category
fetch first 100 rows only;
```

TPC-DS Query 23

```
with frequent_ss_items as
  (select
     substr(i_item_desc, 1, 30) itemdesc, i_item_sk item_sk, d_date solddate, count(*) cnt
   from
     store_sales, date_dim,  item
   where
     ss_sold_date_sk = d_date_sk and  ss_item_sk = i_item_sk and
     d_year in (1999, 1999 + 1, 1999 + 2, 1999 + 3)
   group by
     substr(i_item_desc, 1, 30), i_item_sk, d_date
   having
     count(*) >4
  ),
  max_store_sales as
```

```
  (select
     max(csales) tpcds_cmax
   from
     (select
        c_customer_sk, sum(ss_quantity*ss_sales_price) csales
      from
        store_sales, customer, date_dim
      where
        ss_customer_sk = c_customer_sk and ss_sold_date_sk = d_date_sk and
        d_year in (1999, 1999+1, 1999+2, 1999+3)
      group by
        c_customer_sk
     )
  ),
  best_ss_customer as
  (select
     c_customer_sk, sum(ss_quantity*ss_sales_price) ssales
   from
     store_sales, customer
   where
     ss_customer_sk = c_customer_sk
   group by
     c_customer_sk
   having
     sum(ss_quantity*ss_sales_price) > (95/100.0) *
     (select
        *
      from
        max_store_sales
     )
  )
select
  c_last_name, c_first_name, sales
from
  (select
     c_last_name, c_first_name, sum(cs_quantity*cs_list_price) sales
   from
```

```
  catalog_sales, customer, date_dim
where
  d_year = 1999 and  d_moy = 5 and cs_sold_date_sk = d_date_sk and cs_item_sk
  in
  (select
     item_sk
   from
     frequent_ss_items
  ) and
  cs_bill_customer_sk in
  (select
     c_customer_sk
   from
     best_ss_customer
  ) and
  cs_bill_customer_sk = c_customer_sk
group by
  c_last_name, c_first_name
union all
select
  c_last_name, c_first_name, sum(ws_quantity*ws_list_price) sales
from
  web_sales, customer, date_dim
where
  d_year = 1999 and d_moy = 5 and ws_sold_date_sk = d_date_sk and
  ws_item_sk in
  (select
     item_sk
   from
     frequent_ss_items
  ) and
  ws_bill_customer_sk in
  (select
     c_customer_sk
   from
     best_ss_customer
  ) and
```

```
    ws_bill_customer_sk = c_customer_sk
  group by
    c_last_name, c_first_name
  )
order by
  c_last_name, c_first_name, sales
fetch first 100 rows only;
```

TPC-DS Query 27

```
select
  i_item_id, s_state, grouping(s_state) g_state, avg(cast(ss_quantity as double)) agg1,
  avg(ss_list_price) agg2, avg(ss_coupon_amt) agg3, avg(ss_sales_price) agg4
from
  store_sales, customer_demographics, date_dim, store, item
where
  ss_sold_date_sk = d_date_sk and  ss_item_sk = i_item_sk and
  ss_store_sk = s_store_sk and ss_cdemo_sk = cd_demo_sk and
  cd_gender = 'M' and cd_marital_status = 'U' and cd_education_status = 'College' and
  d_year = 1999 and s_state in ('OH', 'GA', 'LA', 'NM', 'LA', 'MI')
group by
  rollup (i_item_id, s_state)
order by
  i_item_id, s_state
fetch first 100 rows only;
```

TPC-DS Query 36

```
select
  sum(ss_net_profit)/sum(ss_ext_sales_price) as gross_margin, i_category, i_class,
  grouping(i_category)+grouping(i_class) as lochierarchy,
  rank() over (partition by grouping(i_category)+grouping(i_class),
               case
               when grouping(i_class) = 0
```

```
              then i_category end
              order by
                sum(ss_net_profit)/sum(ss_ext_sales_price) asc) as rank_within_parent
from
  store_sales, date_dim d1, item, store
where
  d1.d_year = 2000 and d1.d_date_sk = ss_sold_date_sk and i_item_sk = ss_item_sk and
  s_store_sk = ss_store_sk and s_state in ('MO', 'GA', 'PA', 'AL', 'AL', 'LA', 'MI', 'SC')
group by
  rollup(i_category, i_class)
order by
  lochierarchy desc,
case
when lochierarchy = 0
then i_category end,
     rank_within_parent
fetch first 100 rows only;
```

TPC-DS Query 38

```
select
  count(*)
from
  (select
     distinct c_last_name, c_first_name, d_date
   from
     store_sales, date_dim, customer
   where
     store_sales.ss_sold_date_sk = date_dim.d_date_sk and
     store_sales.ss_customer_sk = customer.c_customer_sk and
     d_month_seq between 1197 and 1197 + 11
   intersect
   select
     distinct c_last_name, c_first_name, d_date
   from
     catalog_sales, date_dim, customer
```

```
   where
     catalog_sales.cs_sold_date_sk = date_dim.d_date_sk and
     catalog_sales.cs_bill_customer_sk = customer.c_customer_sk and
     d_month_seq between 1197 and 1197 + 11
   intersect
   select
     distinct c_last_name, c_first_name, d_date
   from
     web_sales, date_dim, customer
   where
     web_sales.ws_sold_date_sk = date_dim.d_date_sk and
     web_sales.ws_bill_customer_sk = customer.c_customer_sk and
     d_month_seq between 1197 and 1197 + 11
  ) hot_cust
fetch first 100 rows only;
```

TPC-DS Query 43

```
select
  s_store_name, s_store_id,
  sum(
      case
      when (d_day_name='Sunday')
      then ss_sales_price
      else null end) sun_sales,
  sum(
      case
      when (d_day_name='Monday')
      then ss_sales_price
      else null end) mon_sales,
  sum(
      case
      when (d_day_name='Tuesday')
      then ss_sales_price
      else null end) tue_sales,
  sum(
```

```
      case
      when (d_day_name='Wednesday')
      then ss_sales_price
      else null end) wed_sales,
  sum(
      case
      when (d_day_name='Thursday')
      then ss_sales_price
      else null end) thu_sales,
  sum(
      case
      when (d_day_name='Friday')
      then ss_sales_price
      else null end) fri_sales,
  sum(
      case
      when (d_day_name='Saturday')
      then ss_sales_price
      else null end) sat_sales
from
  date_dim, store_sales, store
where
  d_date_sk = ss_sold_date_sk and  s_store_sk = ss_store_sk and s_gmt_offset = -6 and
  d_year = 2002
group by
  s_store_name, s_store_id
order by
  s_store_name, s_store_id, sun_sales, mon_sales, tue_sales,
  wed_sales, thu_sales, fri_sales, sat_sales
fetch first 100 rows only;
```

TPC-DS Query 51

```
WITH web_v1 as
  (select
     ws_item_sk item_sk, d_date,
```

```
    sum(sum(ws_sales_price)) over (partition by ws_item_sk order by
                                   d_date rows between unbounded preceding and current row) cume_sales
   from
    web_sales, date_dim
   where
    ws_sold_date_sk=d_date_sk and  d_month_seq between 1200 and 1200+11 and ws_item_sk is not NULL
   group by
    ws_item_sk, d_date
  ),
  store_v1 as
  (select
    ss_item_sk item_sk, d_date,
    sum(sum(ss_sales_price)) over (partition by ss_item_sk order by
                                  d_date rows between unbounded preceding and current row) cume_sales
   from
    store_sales, date_dim
   where
    ss_sold_date_sk=d_date_sk and d_month_seq between 1200 and 1200+11 and ss_item_sk is not NULL
   group by
    ss_item_sk, d_date
  )
select
  *
from
  (select
    item_sk, d_date, web_sales, store_sales,
    max(web_sales) over (partition by item_sk order by
                         d_date rows between unbounded preceding and current row) web_cumulative,
    max(store_sales) over (partition by item_sk order by
                         d_date rows between unbounded preceding and current row) store_cumulative
   from
    (select
     case
     when web.item_sk is not null
     then web.item_sk
     else store.item_sk end item_sk,
     case
```

```
      when web.d_date is not null
      then web.d_date
      else store.d_date end d_date,
           web.cume_sales web_sales,
           store.cume_sales store_sales
      from
        web_v1 web
        full outer join store_v1 store
        on (web.item_sk = store.item_sk and
            web.d_date = store.d_date)
     )x
  )y
where
  web_cumulative > store_cumulative
order by
  item_sk, d_date
fetch first 100 rows only;
```

TPC-DS Query 59

```
with wss as
  (select
     d_week_seq, ss_store_sk,
     sum(
         case
         when (d_day_name='Sunday')
         then ss_sales_price
         else null end) sun_sales,
     sum(
         case
         when (d_day_name='Monday')
         then ss_sales_price
         else null end) mon_sales,
     sum(
         case
         when (d_day_name='Tuesday')
```

```
        then ss_sales_price
        else null end) tue_sales,
    sum(
        case
        when (d_day_name='Wednesday')
        then ss_sales_price
        else null end) wed_sales,
    sum(
        case
        when (d_day_name='Thursday')
        then ss_sales_price
        else null end) thu_sales,
    sum(
        case
        when (d_day_name='Friday')
        then ss_sales_price
        else null end) fri_sales,
    sum(
        case
        when (d_day_name='Saturday')
        then ss_sales_price
        else null end) sat_sales
  from
    store_sales, date_dim
  where
    d_date_sk = ss_sold_date_sk
  group by
    d_week_seq, ss_store_sk
 )
select
 s_store_name1, s_store_id1, d_week_seq1, sun_sales1/sun_sales2, mon_sales1/mon_sales2,
 tue_sales1/tue_sales2, wed_sales1/wed_sales2, thu_sales1/thu_sales2,
 fri_sales1/fri_sales2, sat_sales1/sat_sales2
from
 (select
    s_store_name s_store_name1, wss.d_week_seq d_week_seq1, s_store_id s_store_id1,
    sun_sales sun_sales1, mon_sales mon_sales1, tue_sales tue_sales1, wed_sales wed_sales1,
```

```
    thu_sales thu_sales1, fri_sales fri_sales1, sat_sales sat_sales1
   from
    wss, store, date_dim d
   where
    d.d_week_seq = wss.d_week_seq and  ss_store_sk = s_store_sk and
    d_month_seq between 1206 and 1206 + 11
  ) y,
  (select
    s_store_name s_store_name2, wss.d_week_seq d_week_seq2, s_store_id s_store_id2,
    sun_sales sun_sales2, mon_sales mon_sales2, tue_sales tue_sales2,
    wed_sales wed_sales2, thu_sales thu_sales2, fri_sales fri_sales2,
    sat_sales sat_sales2
   from
    wss, store, date_dim d
   where
    d.d_week_seq = wss.d_week_seq and  ss_store_sk = s_store_sk and
    d_month_seq between 1206+ 12 and 1206 + 23
  ) x
where
  s_store_id1=s_store_id2 and d_week_seq1=d_week_seq2-52
order by
  s_store_name1, s_store_id1, d_week_seq1
fetch first 100 rows only;
```

TPC-DS Query 64

```
with cs_ui as
  (select
    cs_item_sk, sum(cs_ext_list_price) as sale,
    sum(cr_refunded_cash+cr_reversed_charge+cr_store_credit) as refund
   from
    catalog_sales, catalog_returns
   where
    cs_item_sk = cr_item_sk and  cs_order_number = cr_order_number
   group by
    cs_item_sk
```

```
 having
   sum(cs_ext_list_price)>2*sum(cr_refunded_cash+cr_reversed_charge+cr_store_credit)
),
cross_sales as
(select
   i_product_name product_name, i_item_sk item_sk, s_store_name store_name,
   s_zip store_zip, ad1.ca_street_number b_street_number,
   ad1.ca_street_name b_street_name,  ad1.ca_city b_city,
   ad1.ca_zip b_zip,  ad2.ca_street_number c_street_number,
   ad2.ca_street_name c_street_name, ad2.ca_city c_city,
   ad2.ca_zip c_zip,  d1.d_year as syear, d2.d_year as fsyear,
   d3.d_year s2year, count(*) cnt, sum(ss_wholesale_cost) s1,  sum(ss_list_price) s2,
   sum(ss_coupon_amt) s3
 FROM
   store_sales, store_returns, cs_ui, date_dim d1, date_dim d2, date_dim d3,
   store, customer, customer_demographics cd1, customer_demographics cd2, promotion,
   household_demographics hd1, household_demographics hd2, customer_address ad1,
   customer_address ad2, income_band ib1, income_band ib2, item
 WHERE
   ss_store_sk = s_store_sk AND ss_sold_date_sk = d1.d_date_sk AND
   ss_customer_sk = c_customer_sk AND ss_cdemo_sk= cd1.cd_demo_sk AND
   ss_hdemo_sk = hd1.hd_demo_sk AND ss_addr_sk = ad1.ca_address_sk and
   ss_item_sk = i_item_sk and ss_item_sk = sr_item_sk and
   ss_ticket_number = sr_ticket_number and ss_item_sk = cs_ui.cs_item_sk and
   c_current_cdemo_sk = cd2.cd_demo_sk AND c_current_hdemo_sk = hd2.hd_demo_sk AND
   c_current_addr_sk = ad2.ca_address_sk and c_first_sales_date_sk = d2.d_date_sk and
   c_first_shipto_date_sk = d3.d_date_sk and ss_promo_sk = p_promo_sk and
   hd1.hd_income_band_sk = ib1.ib_income_band_sk and
   hd2.hd_income_band_sk = ib2.ib_income_band_sk and
   cd1.cd_marital_status <> cd2.cd_marital_status and
   i_color in ('grey', 'peach', 'blue', 'misty', 'lavender', 'magenta') and
   i_current_price between 30 and 30 + 10 and i_current_price between 30 + 1 and 30 + 15
 group by
   i_product_name, i_item_sk, s_store_name, s_zip, ad1.ca_street_number,
   ad1.ca_street_name, ad1.ca_city, ad1.ca_zip, ad2.ca_street_number, ad2.ca_street_name,
   ad2.ca_city, ad2.ca_zip, d1.d_year, d2.d_year, d3.d_year
)
```

```
select
  cs1.product_name, cs1.store_name, cs1.store_zip, cs1.b_street_number, cs1.b_street_name,
  cs1.b_city, cs1.b_zip, cs1.c_street_number, cs1.c_street_name, cs1.c_city, cs1.c_zip,
  cs1.syear, cs1.cnt, cs1.s1 as s11, cs1.s2 as s21, cs1.s3 as s31, cs2.s1 as s12,
  cs2.s2 as s22, cs2.s3 as s32, cs2.syear, cs2.cnt
from
  cross_sales cs1, cross_sales cs2
where
  cs1.item_sk=cs2.item_sk and cs1.syear = 1999 and cs2.syear = 1999 + 1 and
  cs2.cnt <= cs1.cnt and cs1.store_name = cs2.store_name and cs1.store_zip = cs2.store_zip
order by
  cs1.product_name, cs1.store_name, cs2.cnt, cs1.s1, cs2.s1;
```

TPC-DS [26] Query 67

```
select *
from
  (select
     i_category, i_class, i_brand, i_product_name, d_year, d_qoy, d_moy, s_store_id, sumsales,
     rank() over (partition by i_category order by sumsales desc) rk
   from
     (select
        i_category, i_class, i_brand, i_product_name, d_year, d_qoy, d_moy, s_store_id,
        sum(coalesce(ss_sales_price*ss_quantity, 0)) sumsales
      from
        store_sales, date_dim, store, item
      where
        ss_sold_date_sk=d_date_sk and
        ss_item_sk=i_item_sk and
        ss_store_sk = s_store_sk and
        d_month_seq between 1215 and 1215+11
      group by
        rollup(i_category, i_class, i_brand, i_product_name, d_year, d_qoy, d_moy, s_store_id)
     )dw1
  ) dw2
where
```

```
  rk <= 100
order by
  i_category, i_class, i_brand, i_product_name, d_year, d_qoy, d_moy, s_store_id, sumsales, rk
fetch first 100 rows only;
```

TPC-DS Query 74

```
with year_total as
  (select
     c_customer_id customer_id, c_first_name customer_first_name, c_last_name customer_last_name,
     d_year as year, max(ss_net_paid) year_total, 's' sale_type
   from
     customer, store_sales, date_dim
   where
     c_customer_sk = ss_customer_sk and ss_sold_date_sk = d_date_sk and d_year in (2001, 2001+1)
   group by
     c_customer_id, c_first_name, c_last_name, d_year
   union all
   select
     c_customer_id customer_id, c_first_name customer_first_name,
     c_last_name customer_last_name, d_year as year, max(ws_net_paid) year_total, 'w' sale_type
   from
     customer, web_sales, date_dim
   where
     c_customer_sk = ws_bill_customer_sk and ws_sold_date_sk = d_date_sk and
     d_year in (2001, 2001+1)
   group by
     c_customer_id, c_first_name, c_last_name, d_year
  )
select
  t_s_secyear.customer_id, t_s_secyear.customer_first_name, t_s_secyear.customer_last_name
from
  year_total t_s_firstyear, year_total t_s_secyear, year_total t_w_firstyear,
  year_total t_w_secyear
where
  t_s_secyear.customer_id = t_s_firstyear.customer_id and
```

```
  t_s_firstyear.customer_id = t_w_secyear.customer_id and
  t_s_firstyear.customer_id = t_w_firstyear.customer_id and
  t_s_firstyear.sale_type = 's' and t_w_firstyear.sale_type = 'w' and
  t_s_secyear.sale_type = 's' and t_w_secyear.sale_type = 'w' and
  t_s_firstyear.year = 2001 and t_s_secyear.year = 2001+1 and
  t_w_firstyear.year = 2001 and t_w_secyear.year = 2001+1 and
  t_s_firstyear.year_total > 0 and t_w_firstyear.year_total > 0 and
case
when t_w_firstyear.year_total > 0
then t_w_secyear.year_total / t_w_firstyear.year_total
else null end >
case
when t_s_firstyear.year_total > 0
then t_s_secyear.year_total / t_s_firstyear.year_total
else null end
order by
  3, 2, 1
fetch first 100 rows only;
```

TPC-DS Query 97

```
with ssci as
  (select
     ss_customer_sk customer_sk, ss_item_sk item_sk
   from
     store_sales, date_dim
   where
     ss_sold_date_sk = d_date_sk and d_month_seq between 1197 and 1197 + 11
   group by
     ss_customer_sk, ss_item_sk
  ),
  csci as
  (select
     cs_bill_customer_sk customer_sk, cs_item_sk item_sk
   from
     catalog_sales, date_dim
```

```
   where
     cs_sold_date_sk = d_date_sk and d_month_seq between 1197 and 1197 + 11
   group by
     cs_bill_customer_sk, cs_item_sk
  )
select
  sum(
      case
      when ssci.customer_sk is not null and csci.customer_sk is null
      then cast(1 as bigint)
      else cast(0 as bigint) end) store_only,
  sum(
      case
      when ssci.customer_sk is null and
           csci.customer_sk is not null
      then cast(1 as bigint)
      else cast(0 as bigint) end) catalog_only,
  sum(
      case
      when ssci.customer_sk is not null and
           csci.customer_sk is not null
      then cast(1 as bigint)
      else cast(0 as bigint) end) store_and_catalog
from
  ssci
  full outer join csci
  on (ssci.customer_sk=csci.customer_sk and
      ssci.item_sk = csci.item_sk)
fetch first 100 rows only;
```