LayoutPilot: A Lakehouse Physical Design Advisor

Guanli Liu
The University of Melbourne
guanli.liu1@unimelb.edu.au

Andreas Kipf
University of Technology Nuremberg
andreas.kipf@utn.de

Renata Borovica-Gajic
The University of Melbourne
renata.borovica@unimelb.edu.au

Abstract

Physical design has a major impact on query performance in lakehouse systems, especially through partitioning and intra-file layout. A poor physical design can substantially degrade query performance, while it is costly to revise after data ingestion. Yet, existing systems provide little guidance for choosing partitioning and layout configurations. Recent approaches address this issue only after ingestion by triggering automatic re-partitioning and re-layout when the redo cost is lower than the query performance degradation. Consequently, earlier queries may already incur substantial overhead under a suboptimal physical design, while the redesign process itself remains expensive because it requires rewriting data. In this paper, we present LayoutPilot, an interactive workload-aware advisor for single-table lakehouse physical design at ingestion time. Our demonstration shows how users can inspect recommendation evidence, compare design candidates, and validate whether the estimated rankings align with observed query performance.

PVLDB Reference Format:
Guanli Liu, Andreas Kipf, and Renata Borovica-Gajic. LayoutPilot: A Lakehouse Physical Design Advisor. PVLDB, 19(1): XXX-XXX, 2026.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/Liuguanli/layout_advisor.

1 Introduction

Lakehouse [6] has emerged as a major architecture for large-scale analytics, combining warehouse-style data management with the flexibility of data lakes. In practice, this architecture is commonly realized through open table formats such as *Delta Lake* [2], *Apache Iceberg* [4], and *Apache Hudi* [3]. Within these table formats, physical design strongly affects query performance, especially through how data is partitioned across files and organized within files. Since revising these decisions after data ingestion is costly, users often need to choose partition and layout columns early, yet existing systems provide limited support for systematic configuration.

A central aspect of lakehouse physical design is how data is partitioned across files and how records are organized within files. Partition shapes coarse-grained data pruning by determining which subsets of data can be excluded early, while layout influences fine-grained pruning and scan efficiency within the selected data. Therefore, these choices directly affect data skipping, i.e., query performance. Choosing effective partitioning and layout is challenging because the design space is large, attributes can be correlated, and different queries favor different designs.

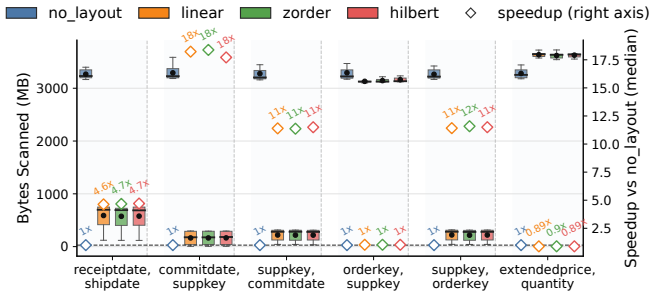


Figure 1: Performance of the query template "`SELECT orderkey FROM lineitem WHERE commitdate BETWEEN ? AND ? AND suppkey BETWEEN ? AND ?`" on TPC-H. Boxplots show bytes scanned and diamond markers indicate speedup relative to the no-layout baseline.

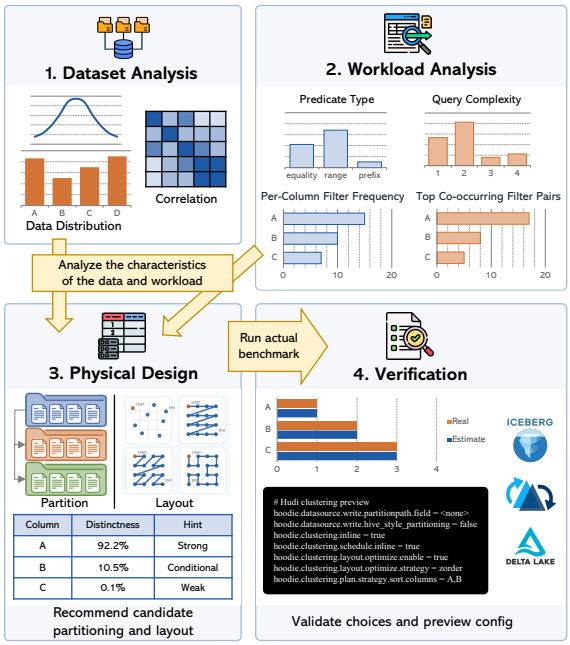
As shown in Figure 1, we evaluate ten range queries over six candidate column groups. Across linear, Z-order [10], and Hilbert [8] layouts, most groups containing `suppkey` can reduce bytes scanned. However, `<orderkey,suppkey>` provides almost no benefit: placing `orderkey` first dominates the physical ordering and weakens the pruning effect of `suppkey`. In contrast, `<commitdate,suppkey>` performs well because `commitdate` is less selective and thus interferes less with the ordering benefit of `suppkey`. Interestingly, `<receiptdate,shipdate>` also performs well, although neither column appears in the predicates, because both are correlated with `commitdate` and still support effective pruning. This example shows that seemingly reasonable layouts can behave very differently. Thus, physical design cannot be chosen reliably by heuristics alone.

Despite their importance, partition and layout remain difficult to configure in practice. Current lakehouse systems largely delegate these decisions to users by exposing physical design knobs, but provide limited guidance on how to choose them well. Apache Hudi, Apache Iceberg, and Delta Lake already expose mechanisms for partitioning, clustering, and layout optimization, while commercial systems, e.g., Databricks [5], Snowflake [9], and Amazon Redshift [1] provide automated physical-design features.

However, these solutions address physical design only after suboptimal configurations have already degraded query performance, rather than proactively supporting design decisions at ingestion time. In contrast, our system helps users understand why certain columns are recommended and compare multiple candidate designs. This allows practitioners to incorporate their own knowledge and make more informed and robust design decisions at ingestion time.

In this paper, we present LayoutPilot, a workload-aware advisor for *single-table* lakehouse physical design. Given a dataset and a target workload, LayoutPilot goes beyond fixed heuristics by analyzing data characteristics and workload signals, including data

Proceedings of the VLDB Endowment, Vol. 19, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX



Column	Distinctness	Hint
A	92.2%	Strong
B	10.5%	Conditional
C	0.1%	Weak

```
# Hudi clustering preview
hoodie.datasource.write.partitionpath.field = <none>
hoodie.datasource.write.hive_style_partitioning = false
hoodie.clustering.inline = true
hoodie.clustering.schedule.inline = true
hoodie.clustering.layout.optimize.enable = true
hoodie.clustering.layout.optimize.strategy = zorder
hoodie.clustering.plan.strategy.sort.columns = A,B
```

Figure 2: System overview

profiles, predicate usage patterns, and correlations among columns, to rank candidate columns for partitioning and layout. LayoutPilot is applicable to lakehouse systems both with and without automatic physical-design support. For systems without such support, it offers a principled way to guide physical design selection. For systems with automatic clustering, it complements existing mechanisms by supporting informed decisions at *initial ingestion time*. This is important because automated physical-design mechanisms are typically invoked only after query performance has degraded, and redesign is performed only when the expected benefit is estimated to exceed the redesign cost. By combining advisor recommendations with practitioner insight, LayoutPilot provides practical guidance for navigating the lakehouse physical design space.

2 System Description

LayoutPilot is designed as an end-to-end system for interactive physical design configuration, taking a dataset and workload as input and producing partitioning and layout configurations as output. LayoutPilot is based on a simple observation: partitioning and layout affect query processing in different ways and should therefore be guided by different signals. Partitioning operates at a coarse granularity by pruning groups of files, making it most suitable for broad filtering predicates and sensitive to factors such as attribute cardinality. Layout, in contrast, operates within partitions and files, making it more relevant to predicates and access patterns that benefit from record ordering and clustering.

As shown in Figure 2, the overall system organizes into four stages: ❶ dataset analysis, ❷ workload analysis, ❸ physical design, and ❹ verification. The first two stages analyze the characteristics of the data and workload. Stage three uses these signals to recommend candidate partitioning and layout configurations, while stage four validates promising choices in the target lakehouse system.

2.1 Dataset Analysis

The purpose of this stage is to extract data-intrinsic structural properties that can influence later physical design choices. Since LayoutPilot has not yet incorporated workload information at this point, the analysis focuses on characteristics of the dataset itself, including value distributions, value ranges, column types, and lightweight distribution sketches. These signals help estimate whether a column is suitable for partitioning and pruning through layout.

A key but often overlooked component of this stage is correlation analysis. We explicitly model correlation because the quality of a multi-column layout depends not only on the utility of individual columns, but also on how columns interact (cf. Figure 1). Strongly associated attributes can preserve useful locality when placed together and also substitute for one another, whereas weakly related combinations tend to dilute locality and reduce the benefit of ordering. We therefore treat correlation as an important design signal for layout recommendation and permutation ranking.

2.2 Workload Analysis

The key idea of workload analysis is that physical design should reflect not only which columns are referenced, but also how they are referenced. For example, a column that appears often in equality predicates plays a different role from a column that appears in selective range predicates, and these two cases should lead to different design decisions.

This is why LayoutPilot extracts predicate types, per-column filter frequency, co-occurring filter pairs, and query complexity statistics. Equality and `IN` predicates are especially important for partitioning, since they align naturally with coarse-grained pruning. By contrast, selective range predicates are more informative for layout, because ordering is most useful when it improves min-max pruning and reduces fine-grained reads. Co-occurrence statistics are also important because multi-column physical design should reflect columns that are frequently used together, rather than columns that are only individually popular.

2.3 Physical Design

LayoutPilot uses the signals extracted from dataset and workload analysis to recommend both partitioning and intra-file layout configurations. Since partitioning and layout improve pruning at different granularities, the system analyzes them separately for recommendation and evaluates them jointly in the final cost model.

Partition Design. LayoutPilot treats partitioning as a coarse-grained pruning mechanism that excludes groups of files before finer-grained access occurs. Accordingly, the system favors columns that are frequently used in equality-like predicates, especially equality and `IN` filters, and that have moderate cardinality rather than extremely high distinctness. Rather than exposing a single opaque score, LayoutPilot summarizes each candidate partition column using two complementary qualitative indicators: a *partition hint* level (***Strong***, ***Conditional***, ***Weak***), which reflects overall suitability as a partition key, and a *partition risk* level (***Low***, ***Medium***, ***High***), which indicates the likelihood of over-partitioning. The *partition hint* is derived from workload frequency, equality-like usage, distinctness, average predicate selectivity, and inferred type, while the *partition risk* serves as a proxy for excessive partition proliferation.

In-Partition Layout Design. After partition candidates are identified, LayoutPilot recommends a shared intra-file layout configuration for records within partitions. The recommendation is guided by signals propagated from the earlier workload analysis stage, including predicate patterns and selectivity-related statistics. Candidate layout columns are prioritized based on their usefulness for ordered access, particularly their range-predicate frequency, selectivity, and association with other candidate columns. To further improve interpretability, columns identified as correlated are highlighted with different colors in the interface, drawing users' attention to relationships that may affect layout quality.

Before selecting layout columns, LayoutPilot presents each candidate with the same core profiling information used for partition, together with layout-oriented summaries such as an *ordering hint* level (e.g., ***Strong***, ***Conditional***, ***Weak***). Internally, the system computes a heuristic score for each column c to support ranking:

$$s_{\text{layout}}(c) = g\big(f_{\text{range}}(c),\ \text{sel}(c),\ \text{qsel}(c),\ \text{card}(c),\ \text{corr}(c)\big), \quad (1)$$

where $f_{\text{range}}(c)$ captures range-predicate usage, $\text{sel}(c)$ and $\text{qsel}(c)$ summarize predicate- and query-level selectivity, $\text{card}(c)$ reflects distinctness, and $\text{corr}(c)$ measures association with other candidate layout columns. Here, $g(\cdot)$ denotes a heuristic combination of these signals used to rank candidate layout columns.

When multiple layout columns are selected, LayoutPilot also evaluates candidate permutations, since layout effectiveness depends not only on which columns are chosen, but also on their order. Columns at the beginning of the ordering have a stronger influence on the physical layout, while the effect of later columns is typically weaker. The system ranks all the candidates using:

$$s_{\text{perm}}(\pi) = \sum_{i=1}^{k} w_i\, s_{\text{layout}}(c_i) + \sum_{i=2}^{k} \text{assoc}(c_{i-1}, c_i), \quad (2)$$

where $\pi = (c_1, \ldots, c_k)$ is a candidate ordering, w_i is a position-dependent weight, and $\text{assoc}(c_{i-1}, c_i)$ measures adjacent-column compatibility.

LayoutPilot also compares different layout types, such as linear ordering, Z-order, and Hilbert-style layouts, depending on the capabilities of the target system. Different layout types are evaluated under a unified scoring framework. For each candidate π, the system first estimates workload-level pruning effectiveness, including average retained-read ratio, workload coverage under meaningful benefit, and worst-case retained reads. These effectiveness signals are then combined with a structural complexity penalty that reflects the relative implementation cost of different layout families.

2.4 Verification

The final stage compares the shortlisted designs selected by LayoutPilot using more concrete evaluation results on a target system. The goal is to check whether the designs preferred by the advisor are also preferred in practice. Compared with a fully automatic approach, we allow practitioners to incorporate their insights that are not visible to LayoutPilot. After verification, users can directly adopt the best-performing validated design for deployment.

In the current prototype, we use Hudi as the default backend for demonstration. In practice, however, this stage is meant to support verification on a user-chosen target system, preferably in a shadow environment over a limited data sample. This avoids the cost of executing full workloads over large production data, while still providing a practical check before deployment.

3 Demonstration Scenario

We demonstrate LayoutPilot through an end-to-end workflow using the TPC-H `lineitem` (scale factor 1) table and a workload of 1,000 queries generated over this table. The demonstration follows the system pipeline from dataset and workload inspection to partition and layout recommendation, and validation.

Figure 3: Dataset analysis view

Dataset Analysis. We begin with the `lineitem` table and inspect its column profiles and pairwise correlations, as shown in Figure 3. The interface summarizes schema information, distinctness, sampled value distributions, and correlation structure. In this dataset, several date attributes, including `shipdate`, `commitdate`, and `receiptdate`, form a strongly correlated group, while `quantity` and `extendedprice` also show strong association. These signals are later used to support layout recommendation.

Figure 4: Workload analysis view

Workload Analysis. Figure 4 shows the workload summary for the 1000 queries. The workload contains both equality and range predicates, with range filters appearing more frequently overall. The per-column filter-frequency view highlights columns such as `shipdate`, `discount`, and `quantity`, while the co-occurrence view reveals common filter pairs such as `discount+suppkey` and `shipdate+shipmode`. These patterns provide the workload evidence used in subsequent partition and layout decisions.

Partition Design. The system then recommends partition columns, as shown in Figure 5. Here, `shipmode` and `returnflag` are selected as the partition columns as they are heavily used in equality-like predicates and have relatively high workload frequency. By contrast,

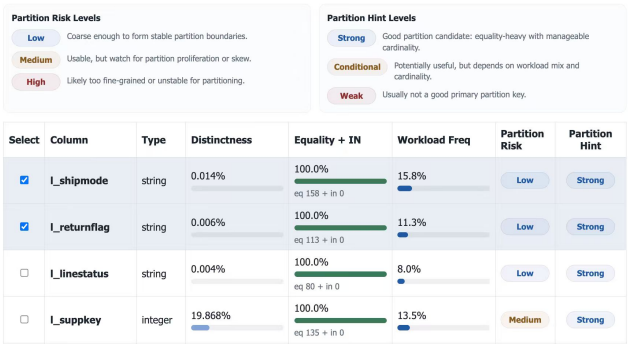


Figure 5: Partition design view

`linestatus` is not selected because it appears less frequently in the workload, despite having similarly low distinctness. suppkey is also excluded because its high distinctness would likely create too many small partitions and files, which is consistent with its medium partition risk indication.

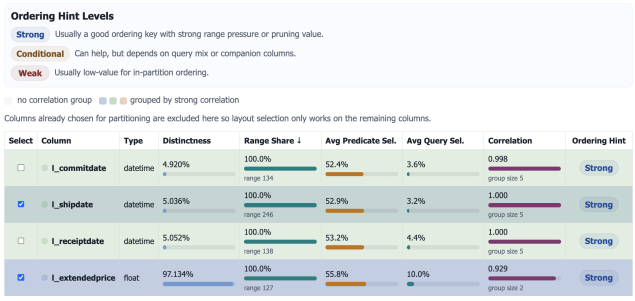


Figure 6: Layout column recommendation view

Layout Recommendation. After partitioning is fixed, LayoutPilot recommends candidate columns for in-partition layout. Figure 6 shows that `extendedprice` and `shipdate` are selected as the main layout candidates. The recommendation is driven by their strong range usage, favorable selectivity, and useful association with other remaining attributes. By contrast, `partkey` is not preferred because its range share, predicate selectivity, and average query selectivity are all low. The view also exposes ordering hints and correlation grouping, helping users understand not only which columns are promising, but also which combinations are more compatible.

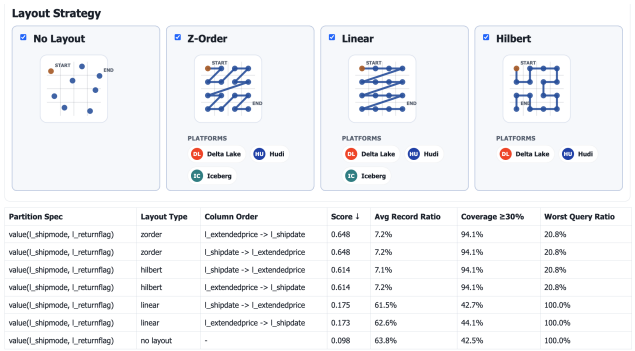


Figure 7: Candidate comparison view

Candidate Comparison. Using the selected partition specification and layout candidates, LayoutPilot generates complete physical design candidates by combining partitioning, layout type, and column order. As shown in Figure 7, the system compares no-layout, linear, Z-order, and Hilbert alternatives under a unified ranking view. In this example, the strongest candidates use the partition specification `<shipmode,returnflag>` together with either Z-order or Hilbert layout over `<extendedprice,shipdate>`. Linear ordering is not competitive, and no-layout ranks last.

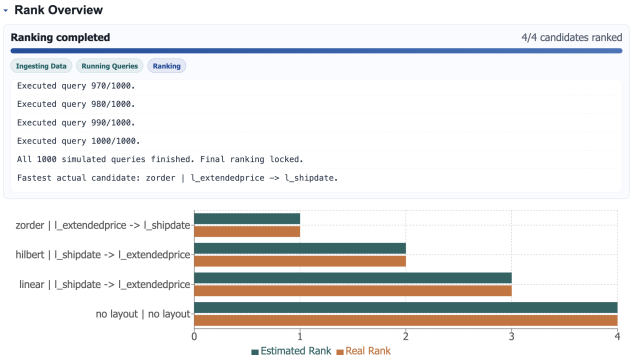


Figure 8: Verification view

Verification. Finally, the shortlisted designs are evaluated in the verification stage. Figure 8 shows that the preferred configurations in this demonstration remain the Z-order and Hilbert layouts over `extendedprice` and `shipdate`. The final validated choice can then be exported as a platform-specific configuration preview for downstream deployment.

4 Conclusion

We presented LayoutPilot, a workload-aware advisor for lakehouse physical design that helps users explore the joint design space of partitioning and intra-file layout. LayoutPilot emphasizes recommendation transparency by helping users understand which attributes are promising, why candidate designs are ranked highly, and how estimated rankings align with observed behavior. Our demonstration shows that interactive physical-design recommendation is both feasible and useful for modern lakehouse settings, reducing manual trial-and-error while providing a practical interface for reasoning about partitioning and layout choices. In future work, we plan to extend LayoutPilot to support joins [7].

References

[1] 2024. *Automated multidimensional data layouts in Amazon Redshift.* https://www.amazon.science/publications/automated-multidimensional-data-layouts-in-amazon-redshift

[2] 2025. *Delta Lake Z Order.* https://delta.io/blog/2023-06-03-delta-lake-z-order/ Accessed: 2025-08-15.

[3] 2026. *Apache Hudi.* https://hudi.apache.org/ Accessed: 2026-03-23.

[4] 2026. *Apache Iceberg.* https://iceberg.apache.org/ Accessed: 2026-03-23.

[5] 2026. *Use liquid clustering for Delta tables.* Accessed: 2026-03-18.

[6] Michael Armbrust, Ali Ghodsi, Reynold S. Xin, and Matei Zaharia. 2021. Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics. In *CIDR*.

[7] Jialin Ding, Umar Farooq Minhas, Badrish Chandramouli, Chi Wang, Yinan Li, Ying Li, Donald Kossmann, Johannes Gehrke, and Tim Kraska. 2021. Instance-Optimized Data Layouts for Cloud Analytics Workloads. In *SIGMOD*. 418–431.

[8] J. K. Lawder and P. J. H. King. 2001. Querying Multi-Dimensional Data Indexed Using the Hilbert Space-Filling Curve. *SIGMOD Record* 30, 1 (March 2001), 19–24. https://doi.org/10.1145/373626.373678

[9] Yipeng Liu, Renfei Zhou, Jiaqi Yan, and Haunchen Zhang. 2026. Workload-Aware Incremental Reclustering in Cloud Data Warehouses. *CoRR* abs/2602.23289 (2026).

[10] Mohamed Ziauddin, Andrew Witkowski, You Jung Kim, Dmitry Potapov, Janaki Lahorani, and Murali Krishna. 2017. Dimensions Based Data Clustering and Zone Maps. *PVLDB* 10, 12 (2017), 1622–1633.