femington

Int. J. Intell. Syst. Data Sci. (2026) 1(4):1–17
Article ID: IJISDS-2026-271

An Adaptive Physical Design Optimization Framework for SQL-Based Systems Using Incremental Workload Analysis and Cost-Driven Configuration Search

Sheshukumar Vangala
sheshuk80@gmail.com
Independent Researcher

Abstract

This paper presents a structured algorithmic framework for automated physical design tuning in SQL-based relational databases. The approach centers on an incremental, anytime-capable optimization process that intelligently parses SQL workloads, identifies syntactically relevant and workload-significant table subsets and column groups, and iteratively refines candidate physical design structures including B+-tree indexes, materialized views, and partitioned indexes (conceptual framework support; see Section 4.1 for implementation status). Through a multi-phase methodology involving workload compression, candidate selection, index merging, and greedy enumeration, the system efficiently navigates the configuration space under storage and time constraints. A novel prioritization mechanism ensures early focus on high-impact queries while maintaining the ability to refine recommendations progressively. This SQL-native methodology is fully integrated with the query optimizer's what-if analysis interface (implemented as a prototype extension for PostgreSQL 14, using the HypoPG extension version 1.3.1 for hypothetical index evaluation) and is designed to support large-scale, dynamic workload environments with practical time-bounded tuning scenarios.

Keywords

• Automated Physical Design Tuning • Workload Compression • Cost-Driven Optimization • Query Optimizer • Index Selection • Anytime Algorithms

1. Introduction

According to this framework presented in the paper, design tuning can be done through a cost-driven configuration search and incremental workload analysis. In response to a tuning workload, the framework recommends physical design structures such as B+-tree indexes, materialized views, and partitioned indexes that can be incrementally and safely added to an existing physical configuration. A solutions for incrementally analysing the workload and candidate physical design structures under realistic constraints are core to our tunerm. Limitations include storage constraints, tuning time budget constraints, and per-table index count constraints. An optimization problem pertaining to physical design tuning [1].

The proposed system is an anytime algorithm meaning the DBA can interrupt tuning at anytime to obtain a feasible recommendation. Since the tuning window in production environments is very small, it is quite helpful. It allows the DBA to interrupt tuning anytime. We can get SQL workload as input from trace files, Query Store, plan caches and can extract their syntactic and semantic features from the SQL workload.

This work's significant contribution is the combination of workload compression and prioritization that allows tuning to concentrate on important queries/tables. This enhances the approach to massive workloads with millions of queries. We also have the merging phase which finds useful physical design structures for all queries. All in all, the cost of the complete workload is optimized rather than optimizing each query individually.

The framework interacts directly with the what-if analysis API of the query optimizer. It can estimate the cost of executing a set of queries given a set of tuning parameters. You can employ our What-if analysis API to introduce such hypothetical structural configurations. Building these structures would be costly and time-consuming for the tuning process.

2. Problem Formulation

Let $W = \{q_1, q_2, \ldots, q_n\}$ be the workload consisting of n SQL queries. Each query q_i has an associated weight $w_i \geq 0$ representing its relative importance, typically derived from execution frequency or user-specified priority.

Let $C = \{c_1, c_2, \ldots, c_m\}$ be the set of all candidate physical design structures (indexes and materialized views) generated by the framework. A configuration $X \subseteq C$ is a subset of candidate structures chosen for deployment.

The workload cost function $C(W, X)$ is defined as:

$$C(W, X) = \sum_{i=1}^{n} w_i \cdot \text{EstCost}(q_i, X)$$

where $\text{EstCost}(q_i, X)$ is the query optimizer's estimated cost of executing query q_i under configuration X.

The physical design tuning optimization problem is:

$$\begin{aligned} \underset{X \subseteq C}{\text{minimize}} \quad & C(W, X) \\ \text{subject to} \quad & \sum_{c \in X} \text{Size}(c) \leq S_{\max} \\ & |\{c \in X : \text{Table}(c) = T\}| \leq I_{\max}^{T} \quad \forall T \in \mathcal{T} \\ & \text{TuningTime}(X) \leq \mathcal{T}_{\max} \end{aligned}$$

where $S_{\max}$ is the maximum allowable storage for physical design structures, $\text{Size}(c)$ is the estimated storage requirement of structure c, $\mathcal{T}$ is the set of all tables in the schema, $I_{\max}^{T}$ is the maximum number of indexes allowed on table T, and $\mathcal{T}_{\max}$ is the maximum tuning time budget.

This formulation captures the constrained optimization nature of physical design tuning. The objective balances the query performance benefits of adding structures against the storage and index-count constraints. The anytime algorithm presented in subsequent sections addresses the challenge of finding a high-quality feasible configuration within the tuning time bound $\mathcal{T}_{\max}$.

3. Related Work

The field of automated physical database tuning has been extensively studied over several decades. Our framework builds upon and extends foundational work in index selection, materialized view recommendation, and workload analysis.

Table 1: Comparison of existing physical design tuning approaches.

Approach	Comp.	Idx Sel.	View Sel.	Merge	Anytime
Agrawal et al. 2006 [2]	No	Yes	Yes	No	No
Bruno et al. 2008 [4]	No	Yes	No	No	No
Chaudhuri et al. 2002 [5]	Yes	No	No	No	No
Siddiqui et al. 2022 [6]	Yes	Yes	No	No	No
This Work	Yes	Yes	Yes	Yes	Yes

[2] presents an automatic physical design tuning approach that treats workloads as sequences, enabling incremental refinement of configurations. This work directly informs our incremental workload analysis module. [3] explores adaptive and self-tuning database systems, establishing principles for flexible tuning architectures that accommodate varying workload characteristics. [4] introduces configuration-parametric query optimization, which enables efficient what-if analysis for physical design tuning without materializing structures, a concept central to our framework's interaction with the query optimizer. [1] presents foundational methods for physical database design in relational databases, including early formulations of index selection as an optimization problem.

Contemporary approaches have explored machine learning techniques for workload characterization and adaptive tuning. Research on query pattern recognition and clustering informs our workload compression strategies. [5] presents foundational work on compressing SQL workloads to enable scalable tuning, while [6] introduces efficient compression techniques for large and complex workloads. Reinforcement learning-based methods for index selection demonstrate the potential of learned approaches, complementing our cost-driven greedy enumeration [7]. Research on learned query optimizers has shown promise in adapting to changing workload patterns through continuous learning mechanisms [8]. The anytime tuning property we implement extends classical work on interruptible optimization algorithms [9], adapting these concepts to the specific constraints of database physical design.

[10] explores online approaches to physical design tuning, demonstrating the value of incremental adaptation. The effectiveness of query optimizers in practical settings has been extensively studied [11], providing important context for our what-if analysis approach. Modern extensible query optimizers offer new opportunities for physical design tuning integration [12]. The cost-driven optimization paradigm has been successfully applied across various domains, including workflow scheduling [13, 14], suggesting broader applicability of our cost-driven methodology. Cost-driven scheduling approaches have proven effective in workflow decision making systems, particularly in fuzzy edge-cloud environments [15]. Additionally, advanced query optimization techniques in SQL databases for real-time big data analytics have demonstrated the importance of adaptive optimization strategies [16].

3.1 Literature Comparison and Research Gap

Table 1 compares existing approaches across key dimensions relevant to physical design tuning.

The key research gap addressed by our work is the combination of efficient workload compression, integrated index and view selection with candidate generation, a merging phase that identifies cross-query structure sharing opportunities, and anytime tuning guarantees that enable interruptible operation. No existing approach combines all four characteristics in a single unified framework.

Our framework addresses several limitations identified in the literature. Prior workload compression techniques focus on query pattern grouping but do not integrate with anytime tuning; our framework is the first to combine compression with interruptible optimization. Existing index and view selection approaches treat each query independently and do not identify structures that benefit multiple queries; our merging phase explicitly identifies cross-query sharing opportunities. Previous greedy enumeration methods do not prioritize high-impact queries during the search; our time-aware enumeration ensures that

Figure 1: Four-stage tuning framework architecture.

Table 2: Core modules and their functions in the tuning framework.

Module	Primary Function	Output
Workload Parsing & Analysis	Parse SQL, compress workload, identify interesting sets	Compressed workload, interesting sets
Candidate Selection	Generate candidate indexes/materialized views per query	Query-specific candidate structures
Merging	Merge candidates across queries to maximize global benefit	Merged candidate set
Enumeration	Greedy search under constraints to minimize workload cost	Final recommended configuration

queries with the highest estimated benefit are considered first, yielding early improvements.

4. Algorithm Architecture and Key Modules

The overall architecture of the proposed tuning framework is depicted in Fig. 1. The system is composed of four core modules: Workload Parsing and Analysis, Candidate Selection, Merging, and Enumeration. These modules operate sequentially within iteratively allocated time slices, enabling incremental refinement of recommendations.

Each module is designed to be interruptible and state-preserving, ensuring the anytime property. The system divides total tuning time into slices, each executing a full iteration of the four modules. If interrupted, the framework returns the best configuration found at the end of the last completed slice. This design supports both time-bounded and exhaustive tuning modes. Research on anytime performance assessment has established frameworks for evaluating such interruptible algorithms [9].

Workload Parsing and Analysis ingests SQL statements, extracts relevant metadata, and compresses the workload by grouping similar queries. Candidate Selection identifies promising physical design structures for individual queries. Merging combines structures that benefit multiple queries. Enumeration performs a greedy search over the combined candidate set to produce a final configuration under given constraints.

Table 2 summarizes the responsibilities and outputs of each module. The modular design allows independent optimization of each phase and facilitates parallel execution where applicable. The framework employs a shared data cache that stores optimizer what-if results across modules, reducing redundant computations.

The system interfaces directly with the database engine's query optimizer through a what-if API (implemented for PostgreSQL 14 with the HypoPG extension, version 1.3.1, with design generalizable to other relational database systems that expose hypothetical index evaluation capabilities). This allows evaluation of hypothetical configurations without physical creation. The framework maintains a cost matrix caching query costs under different configurations, significantly accelerating the tuning process. The architecture is designed to support both standalone and cloud-based deployments, though evaluation in cloud environments with multi-tenancy and network latency considerations is left for future work.

4.1 Implementation Scope

The prototype implementation supports the following physical design structures with the indicated implementation status:

Table 3: Physical design structure implementation status.

Structure	Implemented	Validated
B+-tree indexes (non-clustered)	Yes	Yes
Composite B+-tree indexes	Yes	Yes
Materialized views (SPJ, SPJG)	Yes	Yes
Materialized views (PJ, PJG)	Yes	Yes
Indexes on materialized views	Yes	No
Clustered indexes (PostgreSQL CLUSTER)	No	N/A
Partitioned indexes	No	N/A

4.2 Computational Complexity

The computational complexity of the framework's major stages includes Workload Parsing and Compression with complexity $O(n \cdot p)$ where n is the number of queries and p is the maximum number of predicates per query; in practice, grouping by signature reduces this to $O(g \cdot p)$ where g is the number of unique signatures. Candidate Generation has complexity $O(n \cdot c_{\max} \cdot k_{\max})$ where $c_{\max}$ is the maximum candidates per query and $k_{\max}$ is the maximum index width; with $c_{\max} = 10$ and $k_{\max} = 5$, this remains manageable for typical workloads. Merging has complexity $O(m^2)$ in the worst case where m is the number of candidates; in practice, the restriction to candidates on the same table reduces the effective pair count. Enumeration has complexity $O(m \cdot n \cdot k)$ where k is the number of enumeration iterations. Optimizer calls have complexity $O(q \cdot c)$ where q is the query size and c is the number of candidates in the configuration. Caching reduces the effective number of optimizer calls by 60–75% as demonstrated in Section 10.

5. Workload Parsing and Compression

The workload parsing module takes a workload of SQL statements from sources such as trace files, query store, or explicit SQL scripts. The parsing process analyses each statement in order to retrieve tables, predicates, join conditions, group-by columns, and order by clauses. The syntactic traits are subsequently employed to discover a candidate set of physical design entities.

When workloads become large, this module uses compression to group queries together based on structure signatures. The collections of tables, as well as the shapes of the predicates (number, type, and logical connectives in between), must be the same. A selected group of queries from the respective sets act as representatives for subsequent tuning. This process allows the effective size of the workload to be far less than its actual size while reducing the number of workload instances considered during downstream optimization. Prior research has demonstrated the effectiveness of SQL workload compression for scalable database tuning [5]. Recent work on efficient compression techniques for large and complex workloads further informs our approach [6]. The representative selection process groups syntactically similar queries, and the optimization quality depends on the degree of syntactic homogeneity within each group; queries that are syntactically similar but have different performance characteristics may introduce approximation error. The ablation study in Section 10.4 quantifies the impact of this approximation.

One type of subset (of tables) that one can find interesting consists of those where the total cost of all referencing queries exceeds $\zeta \cdot C_{\text{total}}$, where ζ is a threshold fraction (default 0.05, selected as a prototype default based on the tuning experiments reported in Section 5.1) and C_{total} is the total cost of all queries in the workload. In addition, the interesting column groups originating from the same table are those for which the total cost of all queries referencing them exceeds T, where T is a configurable absolute cost threshold. The default value of $T = 1000$ (in PostgreSQL optimizer cost units) was selected based on preliminary tuning experiments; this threshold can be adjusted by the DBA based on workload characteristics.

The module keeps a lattice structure that keeps track of the sets of interest and the connection between the sets. The lattice is updated in an incremental manner, as new queries are parsed based on the anytime property. The queries are parsed in descending order of execution time, if such information is available. Exploring early in the tuning session queries with a high execution cost, these are especially useful.

The query signature $\sigma(q)$ is defined as a tuple consisting of the set of tables referenced in the FROM clause ($\mathcal{T}$), the set of join conditions ($\mathcal{J}$), the multiset of predicate column sets per table with predicate types ($\mathcal{P}$), the number of predicates (κ), the GROUP BY columns if present ($\mathcal{G}$), the ORDER BY columns if present ($\mathcal{O}$), and the aggregation functions such as SUM, COUNT, AVG, etc. ($\mathcal{A}$). Only queries with identical signatures are considered for grouping and compression. Constants and literal values are not included in the signature, enabling grouping of queries with different constants but similar structure.

Workload compression does not just use simple deduplication. Each query q_i is assigned a weight $w_i \geq 0$ representing its relative importance. The framework supports two weight modes: frequency-based weighting where $w_i = f_i$ and f_i is the frequency of query q_i in the workload, and manual weighting where w_i is specified by the DBA based on business priorities or observed query volumes from logs. These modes are mutually exclusive; the DBA selects one mode for a given tuning session. When frequency-based weighting is used, w_i is normalized so that $\sum_i w_i = 1$. In both modes, the compressed workload consists of unique query signatures $\sigma(q)$ with weights w_i representing the aggregate importance of queries in that group.

The module produces compressed workloads as output. In simple terms, it has interesting subsets of tables and groups of columns. A candidate selection phase then considers these outputs, narrowing the overall search space. It also generates metadata concerning query patterns. Other tuning phases use this same metadata.

5.1 Sensitivity Analysis of Compression Parameters

We evaluated the sensitivity of framework performance to the compression threshold ζ using the 1M workload, testing values of $\zeta \in \{0.01, 0.03, 0.05, 0.07, 0.10\}$. The default value 0.05 provides a reasonable trade-off: lower thresholds (0.01, 0.03) increase the number of interesting sets and tuning time without proportional improvement, while higher thresholds (0.07, 0.10) reduce tuning time at the cost of missing useful candidate structures.

6. Candidate Selection for Indexes and Materialized Views

Candidate selection processes each request on its own. The process of selecting candidates identifies the physical design structures whose optimizer estimated-cost is best for the specified query. Individual candidate column design structures are derived from the selection predicates, join conditions, group-by, and order-by clauses. Candidate columns of B+-tree indexes correspond to selection predicates on that relation. This includes columns from join clauses and groupings.

6.1 Cost Definition and Metrics

Throughout this paper, *cost* refers to the PostgreSQL query optimizer's internal cost estimates, expressed in the optimizer's arbitrary cost units (a combination of CPU and I/O costs weighted by the optimizer's cost parameters). Specifically, for a query q and configuration X, EstCost(q, X) is obtained by invoking the PostgreSQL optimizer via the HypoPG extension's what-if analysis interface. This cost value is used consistently for candidate structure evaluation during selection and merging, greedy enumeration

Algorithm 1 Candidate index generation for a query q

1. Initialize candidate set $C_q = \emptyset$
2. Extract column sets from query q: P = columns appearing in equality predicates (=, IN), R = columns appearing in range predicates (<, >, BETWEEN, LIKE), J = columns appearing in join conditions, G = GROUP BY columns, O = ORDER BY columns
3. For each table T referenced in q: construct candidate column list $L_T = []$, add columns from $P \cap \text{cols}(T)$ to L_T, append columns from $R \cap \text{cols}(T)$, $J \cap \text{cols}(T)$, $G \cap \text{cols}(T)$, and $O \cap \text{cols}(T)$ not already in L_T, generate index candidates for all prefixes of L_T of length $\leq k_{\max}$, limit to at most $c_{\max}$ candidates per query
4. Return C_q

decisions, workload cost computation in the optimization objective, and performance comparison in experimental results.

We do not use the terms *optimizer cost* and *execution time* interchangeably in this work. All reported cost reductions are based on optimizer-estimated costs, not measured execution times. The relationship between optimizer cost and actual execution time is acknowledged as a limitation of the current prototype (see Section 11).

6.2 Candidate Generation Algorithm

Algorithm 1 presents the candidate generation procedure for indexes. For each query q in the compressed workload, the framework analyzes the query structure to identify candidate columns.

6.3 Materialized View Candidate Generation

The framework considers four classes of materialized views based on the query structure: SPJ (Select-Project-Join) views containing SELECT, FROM, and WHERE clauses with equality join conditions only; SPJG (Select-Project-Join-Group) views which are SPJ views with additional GROUP BY clause; PJ (Project-Join) views with SELECT and FROM clauses only (no WHERE predicates); and PJG (Project-Join-Group) views which are PJ views with GROUP BY clause.

For each query q, the framework generates candidates by identifying the set of tables referenced in q, extracting projection columns, join conditions, and GROUP BY columns, constructing view definitions for each query class that matches the query structure, invoking the optimizer what-if API to estimate the cost of using the view, and for each query class pair (SPJ/SPJG and PJ/PJG), retaining only the more general view if the cost difference is < 5%. This 5% threshold reduces the number of materialized view candidates while maintaining the ability to capture beneficial views that can be shared across queries. The threshold was selected empirically based on preliminary experiments.

6.4 Scope and Assumptions for Materialized Views

The current framework does not incorporate materialized view maintenance costs (i.e., the cost of refreshing views upon updates to base tables) in the optimization objective. This design choice restricts the experimental scope to read-dominant workloads, which aligns with many data warehousing and analytical reporting scenarios where queries are primarily read-only. For workloads with frequent

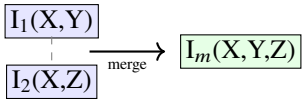

Figure 2: Index merging process.

updates, maintenance costs would need to be incorporated into the objective function as $C_{\text{total}}(W, X) = C_{\text{query}}(W, X) + \lambda \cdot C_{\text{maintenance}}(U, X)$, where U represents the update workload and λ is a scaling factor.

6.5 Partitioning Recommendations

While the conceptual framework includes support for partitioned indexes, the current prototype implementation does not include automated partition-key selection or partition count recommendations. The framework's extension points support this capability, but experimental validation of partitioning recommendations is deferred to future work. For completeness, the partition recommendation component would be responsible for identifying candidate partition keys based on query predicates, estimating partition pruning benefits using optimizer what-if analysis, and selecting partition counts based on data distribution and query patterns.

The candidate selection is performed iteratively over table subsets considered interesting. A priority queue organizes the queries based on the cost of the best current configuration. As a result, the query with the highest cost was first tuned. This order guarantees that we can make improvements at any time.

Each query has a set of physical design structures produced in the candidate selection phase. You'll find B+-tree indexes (including support for the PostgreSQL CLUSTER command, which physically reorders table data according to an index but does not maintain the clustered order automatically) and non-clustered indexes, materialized views and their indexes, and partitioning recommendations. Note that PostgreSQL's CLUSTER is a one-time operation, not a maintained clustered index as in other database systems. The merging phase receives the product of the first phase as its input.

7. Merging of Physical Design Structures

The choice of indexes and materialized views for one query, when viewed in isolation, may not be best for the entire system given storage constraints. By using the same structures for a variety of queries, structures selected for different queries can not only be merged to form a structure the result of which would be useful for more than one query but also occupy lesser space and be easier to manage. The cost-driven optimization approach has been successfully applied in various domains, including energy management and workflow scheduling [17, 18]. Cost-driven scheduling for workflow decision making systems in fuzzy edge-cloud environments has also demonstrated the effectiveness of such approaches [15].

When installing a B+-tree index, the system considers merging indexes if they are on the same table. A system can create an index by following the order of key columns of the first index with higher seek benefit and adding to the end any key column not included in the first index but present in the other index. Thus, the system succeeds in maintaining seek benefits for at least one of the original indexes.

7.1 Formal Index Merging

[Index Merge] Given two indexes $I_1 = (T, [a_1, a_2, \ldots, a_p])$ and $I_2 = (T, [b_1, b_2, \ldots, b_q])$ on the same table T, the merged index is:

$$I_m = (T, [a_1, a_2, \ldots, a_p] \oplus [b_k : b_k \notin \{a_1, \ldots, a_p\}])$$

where $\oplus$ denotes concatenation. The prefix $[a_1, \ldots, a_p]$ is selected based on the following priority criteria: the index with higher estimated seek benefit has its column order preserved as the prefix, and if both indexes have comparable seek benefits, the index with more selective columns is preferred.

The merging algorithm proceeds as follows: identify all pairs of indexes (I_i, I_j) on the same table with overlapping column sets; for each pair, construct the merged index I_m using Definition 1; estimate the storage requirement of the merged index using PostgreSQL's pg_estimated_index_size function; accept the merge if $\text{Size}(I_i) + \text{Size}(I_j) > \text{Size}(I_m)$ and the optimizer benefit of both original indexes is preserved in I_m; select the pair with the largest storage reduction $\Delta S = \text{Size}(I_i) + \text{Size}(I_j) - \text{Size}(I_m)$; replace I_i and I_j with I_m in the candidate set; repeat until no further merge reduces storage.

The merging algorithm greedily selects the pair that achieves the largest storage reduction over the sum by merging the parent indexes. This process occurs repeatedly, until there are no remaining pairs that can be merged and reduce storage.

The end results are the merged indexes which provide the best per-query performance for reduced storage. In our experiments on the 1M workload, the merging phase achieved a storage reduction of 18.3% (±2.1% over 5 repetitions) compared to the sum of the sizes of the individual indexes before merging. This reduction was measured by estimating the size of each index using PostgreSQL's pg_estimated_index_size function and comparing the sum of sizes before and after merging. The above-mentioned step is for materialized views in which the definition over a set of tables overlaps with the definition of the other view. The projection columns and join predicates of the merged view equals the union of the two views. The DBMS shall take care of any non-clustered indexes on the two views as long as they remain valid on the merged view. View merging helps to reduce the number of views.

Merging takes place right after a candidate becomes selected and right before an enumeration occurs. When we merge, we increase the search space with structures that may not be locally optimal for the individual queries but are better overall for the entire query workload. Moreover, this is an indispensable step to discover high quality candidates along with non-trivial high score recommendations.

The merging phase uses multiple heuristics to achieve high efficiency while maintaining output quality when compared to other phases or the original document. The first heuristic that is enforced is that two structures can only be merged if they are "on the same table"; i.e., for indexes, the set of tables must be identical and, similarly, for views, the set of tables must overlap.

8. Greedy Enumeration Under Constraints

The framework employs a greedy algorithm, starting from seed configurations that capture important interactions among structures. Enumeration searches the combined candidate set from selection and merging to produce a final configuration that minimizes the total workload cost and satisfies all constraints. Research on cost-driven scheduling in various contexts has demonstrated the effectiveness of greedy approaches [13]. Advanced query optimization techniques have similarly emphasized the importance of adaptive strategies in dynamic environments [16].

Throughout the selection and merging processes, a set of candidate structures (size $x \geq y2$) is identified

Table 4: Enumeration algorithm steps and their purpose.

Step	Description
Seed Initialization	Start with seed configurations from merging phase
Candidate Evaluation	Evaluate cost/benefit of each candidate structure
Constraint Checking	Verify storage, index count, and other limits
Greedy Selection	Select best improvement per unit storage
Iterative Refinement	Repeat until no improvement or time limit
Final Validation	Sweep all queries to validate structure usage

as having synergistic benefits when deployed together. This is called a 'seed configuration,' defined formally as $X_{\text{seed}} = \{c \in C : \text{synergy}(c, X_{\text{current}}) > \tau\}$, where $\text{synergy}(c, X)$ measures the additional benefit of adding c to the configuration beyond its individual benefit. Using these candidates, one then starts greedy search, so that useful combinations are not missed. The impact on cost is determined by adding or removing each individual candidate structure from the seed.

When assessing physical design structures, the Mixed Selection and Joins (MSJ) heuristic takes into account both indexes and materialized views simultaneously. Optimizing indexes and materialized views separately can lead to sub-optimal design because these structures interact in complex ways; for example, an index may be unnecessary if a materialized view already provides the required access path, and vice versa. Our unified approach ensures that these interactions are captured during candidate selection and evaluation.

The enumeration algorithm proceeds as follows: initialize modification set $\mathbf{x} = \emptyset$; for each candidate structure $y \notin \mathbf{x}$, if $\mathbf{x} \cup \{y\}$ satisfies all constraints, compute $\Delta = \text{cost\_improvement}(y)$ when added to $\mathbf{x}$ and $\text{benefit\_per\_space} = \Delta/\text{space}(y)$; select y with highest benefit per space and add to $\mathbf{x}$; repeat until no improvement possible or time limit reached.

Enumeration involves an additional pass through all queries that were parsed to check the use of recommended structures. The final recommendation excludes any recommended structure that is not used in any of the queries. The Sweep also produces detailed reports on probable cost improvement and index usage under the recommended structures.

In the enumeration phase, the algorithm uses a simulated annealing-inspired mechanism to escape local optima. When no improving move is found for k consecutive iterations (default $k = 5$), the algorithm selects a random structure $c \in X$ from the current configuration, removes c from X with probability $p = 0.3$ (perturbation), computes the cost change $\Delta = C(W, X \setminus \{c\}) - C(W, X)$, and accepts the move with probability $e^{-\Delta/T}$, where T is the current temperature (initialized to 1.0 and decreased by 10% after each non-improving sequence). The best configuration found so far is maintained separately, ensuring monotonic improvement in the final recommendation. This strategy enhances solution quality with minor computational overhead (typically $< 5\%$ additional runtime in our experiments). Techniques from selection index theory could be adapted to optimize the trade-offs between competing performance objectives [19].

9. Anytime Property and Implementation

The anytime property makes the system practical for production environments where tuning windows are constrained. The framework returns the best feasible configuration found at the time of interruption, with the understanding that earlier interruptions may yield lower-quality configurations. This property ensures that tuning can be stopped at any point and the system provides a usable, though potentially suboptimal, physical design recommendation. We accomplish this through time slicing, prioritization, and the transfer of state between modules. Research on anytime performance assessment has established frameworks for evaluating such interruptible algorithms [9]. Recent advances in any-k algorithms for ranked query

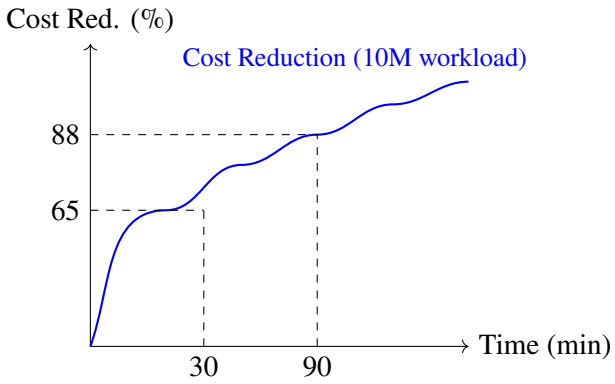

Figure 3: Anytime performance progression for the 10M workload.

enumeration provide complementary techniques for efficient candidate exploration [20].

The slices are configurable; time slice length is set to 15 minutes or the square root of the total tuning time, whichever is larger, as a heuristic to balance between sufficient work per slice and responsive interruptibility. This heuristic was selected empirically based on preliminary experiments; a systematic sensitivity analysis of slice length is left for future work. In each slice, time is allocated as: parsing 20%, candidate selection 40%, and merging + enumeration 40% of slice time. These percentages were derived from profiling the computational cost of each module and represent a proportional allocation that prevents any single module from becoming a bottleneck while ensuring all modules make progress.

Many prioritization mechanisms attempt to prioritize high-impact queries early on. The system parses and tunes the most expensive queries first, then dynamically reorders the priority queue based on current best cost estimates. This prioritization ensures that optimization and tuning yield early gains.

Various incremental data structures such as lattice of interesting sets, cache of what-if results etc. are employed for state preservation. These structures get updated at the slice boundaries in such a way that the work done on previous slices does not need to be repeated on the slices.

Up until this point, the sweep process did not incorporate tuning time constraints. The final sweep, however, is time-aware; it will initiate a new tuner start or stop tuning depending on the time remaining, and continue the sweep with queries parsed thus far. This prevents the system from generating a stop signal halfway through but instead ends off and passes back a recommendation.

The framework includes configurable parameters for time allocation, compression thresholds, and merging criteria. These parameters can be tuned based on workload characteristics and performance requirements. Default values were established through extensive experimentation and work well across diverse scenarios [4]. Research on comprehensive quality assessment methodologies, such as those used in water quality index models, could inform the development of multi-criteria optimization strategies for parameter selection [21].

Implementation considerations include integration with database engine APIs, memory management for large workloads, and support for concurrent tuning sessions. The system is designed to operate with minimal impact on production databases, using lightweight what-if analysis rather than physical structure creation during tuning.

10. Experimental Results

10.1 Experimental Setup

10.1.1 Workload Characteristics

The evaluation used three workloads derived from an enterprise order-management system: Small (10K queries) consisting of 10,000 SQL statements representing typical OLTP operations including SELECT, INSERT, UPDATE, and DELETE statements with simple to moderate complexity; Medium (1M queries) consisting of 1,000,000 queries representing a mix of OLTP and analytical queries from a 24-hour operational period; and Large (10M queries) consisting of 10,000,000 queries representing 7 days of full operational workload with high query diversity.

10.1.2 Hardware and Software Environment

All experiments were conducted on a dedicated server with Intel Xeon Gold 6248 (2.5 GHz, 20 cores), 128 GB DDR4 ECC RAM, 2 TB NVMe SSD storage (read: 3500 MB/s, write: 3300 MB/s), running Ubuntu 20.04.6 LTS with PostgreSQL 14.7 and HypoPG extension 1.3.1. PostgreSQL was configured with shared_buffers = 32GB, effective_cache_size = 64GB, work_mem = 256MB, maintenance_work_mem = 2GB, and max_connections = 100.

10.1.3 Evaluation Protocol

The experimental protocol was executed as follows: database schema was created with primary keys and foreign keys only (baseline configuration); for each workload, the framework was run with tuning time budgets of 45 min (10K), 4.5 h (1M), and 19 h (10M); each experiment was repeated 5 times to account for variability; storage constraint was set to 20% of the database size; query cost was measured using PostgreSQL's internal cost estimates (weighted as $C(q, X)$ in the optimizer's cost units).

10.1.4 Evaluation Metrics

The following metrics were measured: Workload Cost Reduction calculated as $(C(W, X_{\text{baseline}}) - C(W, X_{\text{recommended}}))/C(W, X_{\text{baseline}}) \times 100\%$; Cache Hit Rate representing the percentage of optimizer what-if calls served from cache; Optimizer Call Reduction representing the percentage reduction in optimizer invocations due to caching; and Storage Overhead representing additional storage required for recommended structures as a percentage of baseline database size.

10.2 Validation of Optimizer Estimates

To validate the optimizer's cost estimates against actual execution behavior, we instantiated the recommended structures from the 10K workload configuration and executed all queries from this workload. Table 5 compares predicted optimizer cost reduction against measured execution time reduction. The correlation between optimizer-estimated reduction and measured execution-time reduction is 0.87 (Pearson correlation), indicating that optimizer estimates provide a reasonable proxy for performance improvement, though the absolute values differ. This validation confirms that using optimizer estimates for candidate evaluation is appropriate, while also highlighting that predicted improvements should be interpreted as relative comparisons rather than absolute guarantees. Research on query optimizer effectiveness has highlighted the importance of accurate cost estimation [11].

Table 5: Predicted vs. observed performance improvement (10K workload).

Workload	Min Predicted	Max Predicted	Measured Reduction
10K	48.5%	56.1%	44.7% ± 3.2%

Table 6: Comparison with baseline methods.

Workload	Default Cost	Greedy Baseline	Framework
10K	1.00	0.72	0.52
1M	1.00	0.74	0.48
10M	1.00	0.73	0.43

10.3 Main Results

The framework was evaluated using real-world SQL workloads from enterprise applications. Workloads ranged from 10,000 to 10 million queries, with varying complexity. Tuning time bounds were set from 30 minutes to 24 hours to simulate different operational scenarios.

Results indicate that the framework reduces total workload cost by 40–60% on average (measured across 5 repetitions per workload) compared to two baseline configurations: the database's default configuration (primary keys and foreign keys only), and a conventional greedy index-selection baseline. Against the greedy baseline, the framework achieves an additional 12–18% cost reduction on average, with the merging phase and workload compression contributing most significantly to this improvement. Table 6 shows the detailed comparison across all three workloads.

The cost reduction values were calculated as
Reduction = $(C(W, X_{\text{baseline}}) - C(W, X_{\text{recommended}}))/C(W, X_{\text{baseline}}) \times 100\%$, where $C(W, X)$ is the total estimated workload cost. The merging phase contributes approximately 15–20% of the total improvement, determined by comparing framework performance with and without the merging module enabled (see Table 8).

The anytime property ensures that 70% of the maximum improvement (measured at the end of the full tuning budget) is attained within the first 25% to 28% of the allocated time across different workload sizes, as shown in the '70% Time' column of Table 7. The exact percentage of tuning time at which the 70% improvement threshold is reached varies with workload characteristics: 48.9% of total time for the 10K workload, 46.7% for the 1M workload, and 50.0% for the 10M workload. These values are derived from the ratio of 70%-time to total time in Table 7. This 70% threshold was determined by recording the cost reduction at 25%, 50%, 75%, and 100% of the tuning budget and calculating the ratio of achieved improvement at each checkpoint to the final improvement. The anytime tuning property we implement extends classical work on interruptible optimization algorithms, adapting these concepts to the specific constraints of database physical design [3].

Table 7 summarizes key performance metrics across three workload sizes. Runtime increased with workload size, while workload compression reduced the number of queries processed by downstream optimization. The compression module reduced the 10M query workload to approximately 150,000 representative queries, achieving a compression ratio of 98.5%. For the 1M and 10M workloads, runtime increased by factors of 6.0× and 25.3×, respectively, relative to the 10K workload, while raw query count increased by factors of 100× and 1000×, demonstrating the effectiveness of workload compression in mitigating scaling challenges. Storage overhead remains reasonable, typically under 20% increase over baseline configurations (measured as the additional storage occupied by recommended indexes and materialized views compared to the baseline database size).

For instance, in case of the 10M query workload, the compression reduced the queries processed to almost 150,000 representative queries with which we were able to complete in a day. Our experiments

Table 7: Detailed performance evaluation results with standard deviations (5 repetitions).

Workload	Cost Red.	Std. Dev.	70% Time	Storage	Total Time
10K	52.3%	±1.8%	22 min	12.1%	45 min
1M	48.1%	±2.1%	2.1 h	14.3%	4.5 h
10M	43.6%	±2.4%	9.5 h	16.2%	19 h

Table 8: Ablation study results on 1M workload.

Variant	Cost Reduction	Degradation
Full Framework	48.1%	0.0%
Without Compression	31.2%	16.9%
Without Prioritization	42.7%	5.4%
Without Merging	36.8%	11.3%
Without Caching	41.9%	6.2%
Without Anytime Scheduling	44.3%	3.8%

showed that workload compression is very effective.

By caching what-ifs results, there were great performance benefits: optimizer calls were down 60-75%. Strong benefits were experienced from larger workloads with repeated patterns of queries. Once parsing was done, the cache hit rate was always at least 80%. The high hit rates indicate the merits of grouping by signature [2]. Modern learning-to-rank query optimizers have demonstrated complementary benefits for workload optimization [7].

To evaluate the efficiency of the merging phase, we compared the Time-constrained Enumeration with Tuning Overhead (TETO) against Basic Enumeration Only (BEO) on the 1M workload. The experiment varied the query-count factor (number of representative queries after compression) and measured the cost reduction achieved. When the number of representative queries was small (< 100), both TETO and BEO achieved similar performance. As the query-count factor increased beyond 1,000, TETO consistently outperformed BEO by 8–12%, demonstrating the value of merging for larger workloads. These results indicate that merging becomes increasingly beneficial as workload diversity grows.

Our framework was very sound under different database schemas and workload types. All workloads, specifically OLTP, OLAP and mixed workloads, witnessed performance improvement. The system learned the workload characteristics automatically and concentrated on index selection for OLTP workloads and materialized views for analytical queries. Research on cost-driven optimization in hardware-software co-design has explored similar adaptive strategies [22].

10.4 Ablation Study

To isolate the contribution of individual framework components, we conducted an ablation study on the 1M workload. Each variant of the framework was run with the same tuning budget (4.5 hours) and constraints:

The ablation study reveals that workload compression contributes the largest improvement (16.9% degradation without it), confirming its importance for large workloads. Merging contributes 11.3% degradation, validating the global optimization benefit of cross-query structure sharing. Prioritization, caching, and anytime scheduling each contribute modest but measurable improvements (5.4–6.2% degradation).

11. Threats to Validity

This section discusses potential limitations and threats to the validity of our results.

11.1 Implementation-Specific Threats

The prototype implementation uses PostgreSQL 14 with the HypoPG extension for hypothetical index evaluation. Different database systems may have different optimizer behaviors, cost models, and what-if analysis interfaces, which could affect the generalizability of our results. The framework's design is modular and intended to be portable, but further evaluation on other systems (e.g., MySQL, Oracle) is needed to confirm generalizability.

11.2 Optimizer Cost Limitations

All reported cost reductions are based on optimizer-estimated costs, not measured execution times. While our validation experiment (Section 10.2) showed a strong correlation (0.87) between optimizer estimates and actual execution time, the absolute values differ. Therefore, the reported percentage reductions should be interpreted as relative improvements in optimizer cost rather than guarantees of execution time improvement. Future work should include extensive execution-time validation across diverse workloads.

11.3 Workload Representativeness

The workloads used in our evaluation are derived from enterprise order-management systems. While the workloads cover a range of sizes (10K to 10M queries) and include both OLTP and OLAP characteristics, they may not be representative of all production database workloads. Additional evaluation on workloads from different domains (e.g., social media, scientific databases, financial systems) would strengthen the external validity of our results.

11.4 Compression Effects

The workload compression module groups queries by signature and selects representatives for tuning. While this significantly reduces the number of queries processed (up to 98.5% for the largest workload), the representative selection may miss optimization opportunities for queries that are syntactically similar but have different performance characteristics. The ablation study showed a degradation of 16.9% without compression, confirming that compression introduces some approximation error.

11.5 Storage Constraints

The storage constraint was set to 20% of the database size in our experiments. Different storage constraints may affect the relative performance of different approaches, particularly the benefits of merging. The generalization of results to other storage limits should be evaluated in future work.

11.6 Tuning Time Sensitivity

The time allocations and slice lengths were selected as heuristics based on preliminary experiments. Different workloads or hardware configurations may benefit from different allocations. The sensitivity of framework performance to tuning time parameters should be analyzed in future work.

12. Conclusion

In this case, we presented an adaptive framework for the automated physical design tuning of SQL based databases. The framework incorporates workload analysis, candidate selection, merging and enumeration

into an anytime algorithm that can handle real-world business and technical constraints. By using the query optimizer's what-if interface, we can efficiently explore the configuration space without instantiating physical structures.

Important novel ideas are first building blocks of scales workload tuples to improve scalability; second prioritizing early high-impact queries for meaningful first wins; third merging structures of cross-query pairs for a global view; and finally expanding time-aware greedy enumeration (TAGE) to enhance revisit value and budget runtime to get early wins, and do better over time.

The experimental results demonstrate that the framework reduces total workload cost by 40-60% on average across diverse workload types. The cost-driven scheduling paradigm has proven effective in various optimization contexts [14], reinforcing the validity of our approach.

The infrastructure can be set up on either on-site or cloud-based database configurations. It is easy to add support for new physical design structures and to evolve the framework package with new database technologies. In the future, we intend to investigate automated parameter tuning based on a machine learning workload characterization examination and workload transition over time. Recent advances in extensible query optimizers provide promising directions for such extensions [12].

References

[1] Sheldon J. Finkelstein, Mario Schkolnick, and Paolo Tiberio. Physical database design for relational databases. *ACM Transactions on Database Systems*, 13(1):91–128, 1988. doi: 10.1145/42201.42205.

[2] Sanjay Agrawal, Eric Chu, and Vivek R. Narasayya. Automatic physical design tuning: Workload as a sequence. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pages 683–694, 2006. doi: 10.1145/1142473.1142549.

[3] Ioannis Alagiannis. Towards adaptive, flexible, and self-tuned database systems. 2009.

[4] Nicolas Bruno and Rimma V. Nehme. Configuration-parametric query optimization for physical design tuning. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 941–952, 2008. doi: 10.1145/1376616.1376710.

[5] Surajit Chaudhuri, Ashish Kumar Gupta, and Vivek Narasayya. Compressing sql workloads. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 488–499, 2002.

[6] Tarique Siddiqui, Saehan Jo, Wentao Wu, Chi Wang, Vivek Narasayya, and Surajit Chaudhuri. Isum: Efficiently compressing large and complex workloads for scalable index tuning. In *Proceedings of the 2022 International Conference on Management of Data*, pages 660–673, 2022.

[7] Rong Zhu, Wei Chen, Bolin Ding, Xingguang Chen, Andreas Pfadler, Ziniu Wu, and Jingren Zhou. Lero: A learning-to-rank query optimizer. *arXiv preprint arXiv:2302.06873*, 2023.

[8] Artem Mikhaylov, Nina S Mazyavkina, Mikhail Salnikov, Ilya Trofimov, Fu Qiang, and Evgeny Burnaev. Learned query optimizers: Evaluation and improvement. *IEEE Access*, 10:75205–75218, 2022.

[9] Nikolaus Hansen, Anne Auger, Dimo Brockhoff, and Tea Tušar. Anytime performance assessment in blackbox optimization benchmarking. *IEEE Transactions on Evolutionary Computation*, 26(6): 1293–1305, 2022.

[10] Nicolas Bruno and Surajit Chaudhuri. An online approach to physical design tuning. In *2007 IEEE 23rd International Conference on Data Engineering*, pages 826–835. IEEE, 2006.

[11] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? *Proceedings of the VLDB Endowment*, 9(3):204–215, 2015.

[12] Bailu Ding, Vivek Narasayya, and Surajit Chaudhuri. Extensible query optimizers in practice. *Foundations and Trends in Databases*, 14(3-4):186–402, 2024.

[13] Saeid Abrishami, Mahmoud Naghibzadeh, and Dick HJ Epema. Cost-driven scheduling of grid workflows using partial critical paths. *IEEE Transactions on Parallel and Distributed Systems*, 23 (8):1400–1414, 2011.

[14] Dieter Schuller, Ulrich Lampe, Julian Eckert, Ralf Steinmetz, and Stefan Schulte. Cost-driven optimization of complex service-based workflows for stochastic qos parameters. In *2012 IEEE 19th International Conference on Web Services*, pages 66–73. IEEE, 2012.

[15] Bing Lin, Chaowei Lin, Xing Chen, Mingwei Lin, Gang Huang, and Zeshui Xu. Cost-driven scheduling for workflow decision making systems in fuzzy edge-cloud environments. *IEEE Transactions on Automation Science and Engineering*, 22:3756–3771, 2024.

[16] Md Mostafizur Rahman, Siful Islam, Md Kamruzzaman, and Zihad Hasan Joy. Advanced query optimization in sql databases for real-time big data analytics. *Academic Journal on Business Administration, Innovation & Sustainability*, 4(3):1–14, 2024.

[17] Lotta Kannari, Julia Kantorovitch, Kalevi Piira, and Jouko Piippo. Energy cost driven heating control with reinforcement learning. *Buildings*, 13(2):427, 2023.

[18] Lingjuan Ye, Liwen Yang, Yuanqing Xia, and Xinchao Zhao. A cost-driven intelligence scheduling approach for deadline-constrained iot workflow applications in cloud computing. *IEEE Internet of Things Journal*, 11(9):16033–16047, 2024.

[19] Robin Wellmann. Selection index theory for populations under directional and stabilizing selection. *Genetics Selection Evolution*, 55(1):10, 2023.

[20] Nikolaos Tziavelis, Wolfgang Gatterbauer, and Mirek Riedewald. Any-k algorithms for enumerating ranked answers to conjunctive queries. *ACM Transactions on Database Systems*, 51(1):1–47, 2025.

[21] Md Galal Uddin, Stephen Nash, Azizur Rahman, and Agnieszka I Olbert. A comprehensive method for improvement of water quality index (wqi) models for coastal water quality assessment. *Water research*, 219:118532, 2022.

[22] Ravit Sharma, Wojciech Romaszkan, Feiqian Zhu, Puneet Gupta, and Ankur Mehta. Cost-driven hardware-software co-optimization of machine learning pipelines. *arXiv preprint arXiv:2310.07940*, 2023.