This is Going to Sound Crazy, But What If We Used Large Language Models to Boost Automatic Database Tuning Algorithms By Leveraging Prior History? We Will Find Better Configurations More Quickly Than Retraining From Scratch!

WILLIAM ZHANG, Carnegie Mellon University, USA
WAN SHEN LIM, Carnegie Mellon University, USA
ANDREW PAVLO, Carnegie Mellon University, USA

Tuning database management systems (DBMSs) is challenging due to trillions of possible configurations and evolving workloads. Recent advances in tuning have led to breakthroughs in optimizing over the possible configurations. However, due to their design and inability to leverage query-level historical insights, existing automated tuners struggle to adapt and re-optimize the DBMS when the environment changes (e.g., workload drift, schema transfer).

This paper presents the Booster framework that assists existing tuners in adapting to environment changes (e.g., drift, cross-schema transfer). Booster structures historical artifacts into query-configuration contexts, prompts large language models (LLMs) to suggest configurations for each query based on relevant contexts, and then composes the query-level suggestions into a holistic configuration with beam search. With multiple OLAP workloads, we evaluate Booster's ability to assist different state-of-the-art tuners (e.g., cost-/machine learning-/LLM-based) in adapting to environment changes. By composing recommendations derived from query-level insights, Booster assists tuners in discovering configurations that are up to 74% better and in up to 4.7× less time than the alternative approach of continuing to tune from historical configurations.

CCS Concepts: • **Information systems** → *Autonomous database administration*; *Database utilities and tools*; • **Computing methodologies** → *Machine learning algorithms.*

Additional Key Words and Phrases: Database Tuning; Large Language Models; ML for Data Management

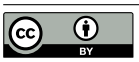
ACM Reference Format:
William Zhang, Wan Shen Lim, and Andrew Pavlo. 2026. This is Going to Sound Crazy, But What If We Used Large Language Models to Boost Automatic Database Tuning Algorithms By Leveraging Prior History? We Will Find Better Configurations More Quickly Than Retraining From Scratch!. *Proc. ACM Manag. Data* 4, 1 (SIGMOD), Article 90 (February 2026), 29 pages. https://doi.org/10.1145/3786704

1 Introduction

Configuring a database management system (DBMS) for modern data-intensive applications has become increasingly difficult for two reasons. First, DBMSs expose trillions of options [52] (e.g., knobs, indexes, query hints). Second, applications constantly evolve, with changes in data access patterns, query types, and load intensities, amongst others [68, 83]. Automated tuners ensure that the DBMS configuration remains optimal as the environment (e.g., workload, data) changes [67].

Optimizing a DBMS for a fixed workload has been well explored by the academic community. Early research focused on heuristics and cost-based search techniques [1, 17, 18, 37, 76]. The last

Authors' Contact Information: William Zhang, wz2@cs.cmu.edu, Carnegie Mellon University , Pittsburgh, Pennsylvania, USA; Wan Shen Lim, wanshenl@cs.cmu.edu, Carnegie Mellon University , Pittsburgh, Pennsylvania, USA; Andrew Pavlo, pavlo@cs.cmu.edu, Carnegie Mellon University , Pittsburgh, Pennsylvania, USA.



This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 2836-6573/2026/2-ART90
https://doi.org/10.1145/3786704

decade saw the rise of using machine learning (ML) to tune single aspects of the DBMS (e.g., knobs [35, 38, 82, 94], indexes [40, 75, 90]) or across all aspects of the DBMS [96]. Recent advances have focused on large language model (LLM)-based methods [29, 35].

Existing research has shown the efficacy of these tuners in finding beneficial configurations for a fixed workload. However, these tuners are unable to adapt to environment changes (e.g., workload drift, different schemas) due to their inherent designs. For example, heuristic tuners [1, 37] cannot incorporate historical knowledge due to their fixed algorithms. ML-tuners [41, 96, 98] struggle as environment changes (e.g., Q1 no longer exists) corrupt their internal representations (e.g., how they represent Q1's tunables). These design issues prevent tuners from effectively exploiting query-level semantics (e.g., query plan) to learn from historical knowledge. Consequently, these tuners take longer to re-optimize after environment changes and end up stuck in subpar configurations.

Given this, we introduce the **Booster** framework that assists an existing tuner (e.g., cost-/ML-/LLM-based) in adapting to environment changes. Booster first organizes historical tuning artifacts into structured insights. When the DBMS environment changes, Booster obtains configurations derived from relevant insights based on each individual query's semantics, composes them into a holistic configuration [96] (i.e., its findings), and injects those findings into the tuner for further refinement. We evaluate Booster's ability to assist state-of-the-art tuners in adapting to new DBMS environments for OLAP workloads. Our experiments with PostgreSQL show that Booster assists tuners in finding configurations that improve DBMS performance by up to 74% in up to 4.7× less time compared to the alternative approach of continuing to tune from history.

We lay out the paper as follows. We provide background into existing tuners' limitations in adapting to environment changes and ML techniques underpinning our method in Sec. 2. We then provide an overview of Booster in Sec. 3, followed by describing its execution phases in Secs. 4 and 5. We then discuss how to integrate the framework with existing tuners in Sec. 6. We evaluate Booster's ability to assist tuners in adapting in Secs. 7 and 8.

2 Background

An autonomous self-driving DBMS [42, 67, 104] optimizes itself without human intervention. Guided by the user's objectives (e.g., minimize runtime), the DBMS optimizes itself for a given workload by deploying *tuners* to find optimal *holistic configurations* [96] that encapsulate all of the DBMS's tunable facets. For instance, a single *holistic configuration* can prescribe system knobs to set, indexes to build, and hints to apply for each query in the workload. To find these holistic configurations, the DBMS's tuners iteratively explore and experiment with different configurations to the best of their individual capabilities. These include cost-based search and ML-based tuners that target individual DBMS aspects (e.g., knobs [1, 38, 82, 94], physical design [8, 24, 40, 75, 90]), holistic tuners that reason across multiple DBMS aspects [96], and large language model (LLM)-based tuners [29, 35, 93]. We first define adaptivity for tuners and then discuss existing tuners along with their challenges in adapting to different deployments.

2.1 Adaptivity for Tuners

As a tuner optimizes DBMSs, it acquires experience that includes the tuner's reasoning, explored configurations, and observed behavior of the DBMS. *Adaptivity* reflects a tuner's ability to reuse prior experience when tuning new scenarios, broadly categorized as *transfer* and *drift*. We will discuss each separately.

Transfer Scenario. This addresses the case where a tuner transfers experiences from past DBMS deployments to a previously unseen deployment. This ranges from cases where the historical and

target deployments are the same to cases where the schemas differ. For example, after tuning a TPC-H [2] instance, the tuner transfers its experiences to optimize a TPC-DS [81] instance.

Drift Scenario. This covers scenarios where a tuner optimizes a DBMS deployment whose characteristics change over time [48, 83, 88]. These include changes to query parameters, query templates, query volume (i.e., load spikes), hardware (e.g., instance upgrades), and underlying data (e.g., `BULK INSERT`). These drifts are common, with Redshift observing 50% of their production clusters have 50% of queries repeating exactly (i.e., same query template and parameters) daily [83].

2.2 Existing Tuning Frameworks

We next provide an overview of existing state-of-the-art tuners, which we group into three categories: (1) heuristic and cost-based, (2) ML-based, and (3) LLM-based.

Heuristic and Cost-based Tuners. These tuners execute a fixed algorithm to explore configurations [1, 11, 18, 37] obtained with heuristics. They then rely on query plan costs via a "what-if" mechanism [17] to guide the search process.

ML-based and Holistic Tuners. These tuners rely on ML techniques to tune the DBMS. While some of these tuners only target specific aspects (e.g., knobs, indexes) of the DBMS [41, 75, 82, 94], holistic tuners [96] reason across all aspects. These tuners rely on an internal model to find beneficial configurations through trial and error. These models capture deployment aspects (e.g., workload, schema) as bits in the models' representation. For instance, a specific bit output by the model may instruct the tuner to build an index [41, 86], set a system knob [82, 94], or apply a query hint [96].

LLM-based Tuners. Recent large language model (LLM)-based tuners rely on off-the-shelf models [29] (e.g., GPT-4o [63]) or models fine-tuned on prior experience [35]. They then instruct the LLM with *prompts*. For instance, a prompt can instruct the LLM to assume the role of a database administrator and output a JSON configuration that optimizes the workload. These tuners sample multiple configurations from the LLM and select the best one.

2.3 Tuner Adaptivity Challenges

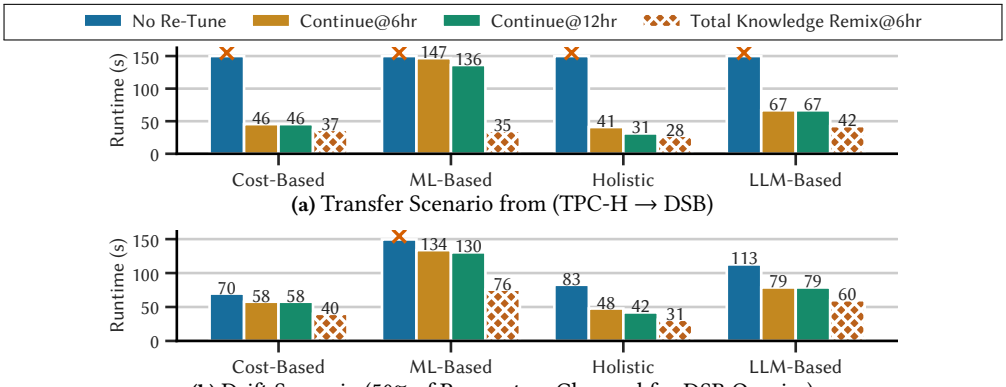
Despite the prevalence of transfer and drift scenarios in practice, existing tuners are unable to adequately adapt to environment changes due to their designs. As we now discuss, their design limitations are due to two core issues:

Rigidity and Brittleness. This issue is present in cost-based and ML-based tuners. For both, identifying relevant experiences is critical to discovering beneficial configurations [51, 82, 97, 99]. Even if relevant experience is accurately identified, problems remain.

Cost-based tuners lack a mechanism to apply prior insights to guide the search. For instance, if prior experience reveals an adverse interaction between two indexes, the tuner should prioritize exploring configurations that exclude those two indexes.

ML-based tuners must re-optimize whenever their internal models' representations change due to the environment (e.g., new query, new indexable column). To compensate for this, researchers proposed using relevant experience to pre-train the internal model [49]. However, pre-training remains infeasible due to potential differences with the target environment and the overhead of re-evaluating historical experiences (i.e., configurations) to obtain performance values for the target schema and workload.

Workload Granularity. Existing tuners adapt on a workload granularity. Rather than tune from scratch, tuners use workload-level telemetry (e.g., DBMS metrics) to identify and start from some historical configuration (i.e., workload mapping [82]). However, this mapping prevents combining



(a) Transfer Scenario from (TPC-H → DSB)

(b) Drift Scenario (50% of Parameters Changed for DSB Queries)

Fig. 1. Tuner Adaptivity Challenges – PostgreSQL runtimes achieved by tuners in transfer and drift scenarios for three configurations: (1) *no re-tune*, (2) *continue* tuning from the best historical configuration for 6hrs and 12hrs, and (3) *remix* prior knowledge and then tune for 6hrs. Each tuner has access to historical artifacts: TPC-H (transfer) and prior DSB (drift).

individual query insights across configurations. For instance, configurations C1 and C2 may optimize different workload queries more effectively. Without a query-level adaptation method based on query semantics (e.g., plan) [14, 49], tuners cannot compose the best aspects of C1 and C2 together.

LLM-based tuners ignore opportunities to augment the prompt with targeted experience to guide the model's suggestion process on a query-by-query basis. For instance, if a query has been tuned before, the tuner could augment the prompt with past attempts (e.g., query hints, indexes) to provide the LLM with further context to generate more effective configurations [19, 47, 51].

To illustrate how these problems hinder DBMS tuners, we ran workloads on PostgreSQL with two scenarios: (1) an experience transfer from TPC-H [2] to DSB [23] and (2) a previously tuned DSB workload undergoing a parameter drift (i.e., 50% of queries have different parameters). We deploy a (1) cost-based tuner [1, 11, 18], (2) ML-based tuner [98], (3) holistic tuner [96], and (4) LLM-based tuner [11, 29]. Each tuner has access to their historical artifacts: TPC-H for transfer scenario and the previous DSB for drift scenario.

As shown in Fig. 1a and Fig. 1b, both scenarios benefit from continuing to tune from the best historical configuration compared to **No Re-Tune**. However, **Continue** falls short of what is achievable by **Total Knowledge Remix**. By fully remixing historical tuning knowledge, tuners discover configurations that are **13–74%** better than those found by **Continue**. To achieve this for a range of tuners, we propose leveraging recent advances in LLMs to reason about individual queries and adapt them to different environments.

2.4 LLM-based Query Adaptation

Recent research in LLMs has shown their capabilities in optimization [29, 51, 80, 93] and schema understanding [79, 91, 100]. However, providing curated information to the LLM remains unsolved [28, 60, 95]. We provide an exposition into incorporating tuning knowledge into LLMs through four techniques: (1) workload-level prompts, (2) workload-level fine-tune, (3) query fine-tune, and (4) combining enriched query-level prompts with a composition mechanism.

Workload-Level Prompts. This technique (**WL-Prompt**) invokes an off-the-shelf LLM with a workload-level prompt that describes the tuning task, the workload, and additional information

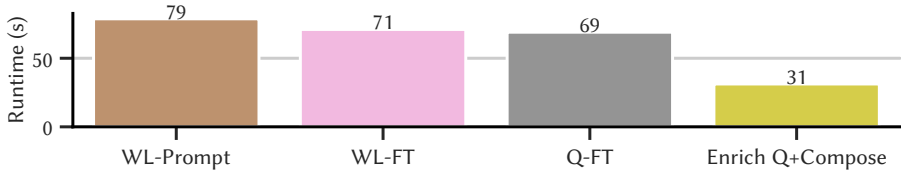


Fig. 2. LLM-based Query Adaptation – DSB workload runtime achieved by prompting an off-the-shelf LLM (**WL-Prompt**), fine-tuning a LLM with historical knowledge at workload (**WL-FT**) and query (**Q-FT**) granularities, and **Enrich Q+Compose** that combines query prompts enriched with historical attempts and a composition mechanism. All techniques utilize experience generated over 12hrs by a holistic tuner tuning a DSB workload where 50% of queries have different parameters.

(e.g., DBMS telemetry, workload summary) as context [47]. This technique utilizes the LLM's innate abilities and pre-trained knowledge to generate configurations rather than historical knowledge.

Workload-Level Fine-Tune. This technique (**WL-FT**) fine-tunes a LLM by using historical experiences to alter the LLM's weights [35]. Similarly to **WL-Prompt**, this technique then invokes the fine-tuned LLM to obtain candidate configurations. By fine-tuning, this technique improves the LLM's ability to generate more effective configurations for the target workload.

Query Fine-Tune. In comparison to **WL-FT**, this technique (**Q-FT**) fine-tunes a LLM on queries. Although this allows the LLM to relate queries to configurations, it requires prompting the LLM with a query prompt. As such, rather than generating workload-level configurations, the LLM generates configurations for each query that are then combined into a holistic configuration [96].

Enriched Query-Level Prompts and Composition. This technique (**Enrich Q+Compose**) is agnostic to whether the LLM is fine-tuned or not. On a per-query basis, this technique enriches the query prompt with prior tuning attempts based on the query (e.g., SQL text, plan) and then instructs the LLM to generate configurations with those historical references [19, 47, 95]. The technique then uses a composition mechanism to combine the query-level configurations into a holistic configuration while resolving conflicts (e.g., different knob values, conflicting indexes). In doing so, this technique achieves **Total Knowledge Remix** by reasoning over all available historical knowledge (e.g., past configurations, Internet).

To understand their efficacy, we first obtain historical experience by using a holistic tuner to optimize a DSB workload for 12hrs. We then evaluate the previous techniques on using this experience to adapt to a drifted workload where 50% of the queries have different parameters. As shown in Fig. 2, **WL-Prompt** performs the worst as it does not use prior experience. **WL-FT** and **Q-FT** have limited improvement over **WL-Prompt** due to issues around granularity and combining query configurations, respectively. In contrast, **Enrich Q+Compose** finds drastically better configurations by exploiting relevant knowledge on a per-query basis and resolving conflicts with a composition mechanism. **Enrich Q+Compose** forms the basis of our method to assist existing tuners in adapting to environment changes by remixing historical knowledge with LLMs.

3 Overview

We present how the Booster framework integrates with an existing tuner (see Sec. 2.2) to improve its ability to adapt to environment changes in Fig. 3. Based on the workload and DBMS, Booster analyzes the repository of artifacts and injects its findings (e.g., start configuration) into the tuner. The injected tuner then further refines the configuration and updates the repository with artifacts.

We next describe how Booster utilizes this repository to derive a holistic configuration [96] (i.e., its findings) for injecting into the tuner. As shown in Fig. 4, Booster's operation is divided into

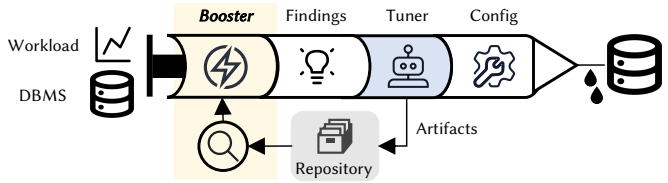


Fig. 3. Booster Overview – The framework integrates with an existing tuner to improve its adaptivity to environment changes. Booster analyzes the artifact repository and injects its findings (e.g., start configuration) into the tuner. The "injected" tuner then refines the configuration and stores its artifacts into the repository.

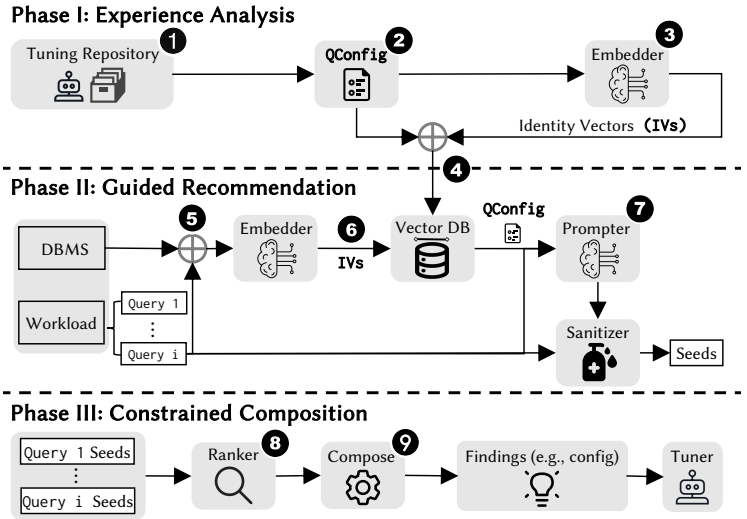


Fig. 4. Booster Architecture – An overview of the framework's three phases. In Phase I, Booster analyzes historical tuning artifacts. In Phase II, Booster generates candidate configurations (i.e., *seeds*) for each query with a LLM. In Phase III, Booster then composes each query's seeds into a holistic configuration that it then provides to the tuner being assisted.

three phases. In Phase I, it monitors the repository for new artifacts generated by the tuner and analyzes them for insights (Sec. 3.1). When the DBMS environment changes, Booster uses these insights to optimize the DBMS. In Phase II, Booster receives the user's target workload and the target DBMS (Sec. 3.2). For each query in the workload, Booster identifies relevant experiences and utilizes them to generate candidate configurations with a LLM. Then in Phase III, Booster combines these query-level candidate configurations with a constrained composition mechanism (Sec. 3.3). Lastly, Booster provides its findings to the tuner to refine. We discuss each in more detail.

3.1 Phase I: Experience Analysis

In this phase, ❶ Booster monitors a repository of tuning experiences. This repository accumulates artifacts generated by a tuner as it optimizes a deployment. Although these artifacts contain substantial information about the DBMS (e.g., workloads, configurations' efficacy), they are not readily searchable and differ across tuners. To standardize these artifacts into a searchable format, Booster first ❷ parses them into query-configuration (QConfig) objects that represent how a query behaves under a specific configuration. Each QConfig includes but is not limited to the relevant schema, the query's execution plan, the configuration (e.g., knobs, indexes), and links to other related QConfigs for structural organization. For instance, Booster can link QConfigs together in chronological order if the QConfigs refer to the same query.

With these QConfigs, Booster next makes them searchable by leveraging recent advances in LLMs' ability to understand tabular data, schemas, and SQL queries [51, 79, 91, 100]. For each QConfig, ❸ Booster utilizes an embedder (e.g., Voyage 3 Large [6]) to generate fixed-length identity vectors (i.e., embeddings [72]) based on text documents derived from the QConfig. For instance, it can derive them by combining the schema and query plan or by combining the anonymized schema and SQL text. Booster combines these identity vectors with the QConfig and ❹ loads them into a vector database for querying in Phase II. We elaborate further in Sec. 4.

3.2 Phase II: Guided Recommendation

At the start of this phase, Booster is now optimizing the DBMS. Booster receives the target workload and a connection to an offline environment for safe exploration [52, 55]. For each query in the workload, ❺ Booster obtains relevant information from the DBMS (e.g., schema, plan) and uses an embedder to generate identity vectors in the same manner as Phase I. With each query's identity vectors, Booster ❻ retrieves relevant historical QConfigs from the vector database, combines them with the target query into the prompt, and ❼ invokes the LLM to recommend configurations based on the QConfigs [19, 47]. As the LLM may suggest invalid configurations (e.g., illegal knob value, invalid index), Booster passes these suggested configurations and the relevant QConfigs through the Sanitizer to obtain configurations for each query that can be deployed. As these configurations target a single query, we refer to them as *query seeds*. We elaborate further in Sec. 4.

3.3 Phase III: Constrained Composition

After obtaining the seeds for each query, Booster then combines them into a holistic configuration [96]. However, seed configurations may specify different parameters (e.g., number of parallel workers, indexes) that conflict when combined. To mitigate this, Booster adopts a two-step process to generate holistic configurations from multiple query seeds. ❽ Booster first ranks all the query seeds and then ❾ runs a beam search algorithm, a variant of best-first search [12, 20, 69], to identify performant holistic configurations. This algorithm terminates when a terminal condition (e.g., elapsed time) is reached. We elaborate on this further in Sec. 5. Finally, Booster injects its discoveries into the tuner being assisted.

4 Experience Analysis & Recommendations

Booster uses QConfigs to guide how its LLM recommends configurations. We now discuss (1) how the framework constructs QConfigs, (2) how it augments the prompt with relevant QConfigs, and (3) how Sanitizer produces seed configurations for each query.

4.1 QConfig Construction

During Phase I, Booster mines the repository of artifacts generated by the tuner. Booster analyzes the configurations explored over time (i.e., *trajectory*) to identify interesting configurations (e.g., ones that improve the user's objective function). On a per-query basis, Booster constructs a QConfig from each interesting configuration. As shown in Fig. 5, this QConfig contains both the DBMS's configuration (e.g., knobs, indexes) and query-specific information (e.g., SQL text, plan). Each QConfig contains a link to the QConfig of the same query that is built from the trajectory's next configuration (i.e., discovered by the tuner chronologically in the future). In Fig. 5's example, Q1-C0 links to Q1-C2, which then links to Q1-C5.

However, Booster cannot directly use these QConfigs for vector-based similarity search [26, 43]. Instead, Booster must first derive a fixed-length vector representation from each QConfig before it can use distance functions (e.g., cosine distance, euclidean distance) for similarity search. Recent research has proposed using embedders (e.g., Voyage 3 Large [6]) for natural language to SQL [26]

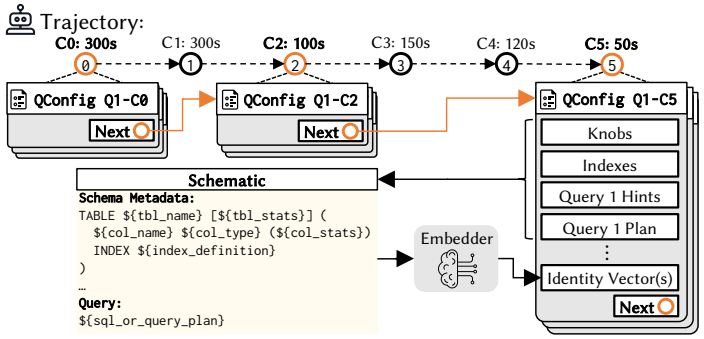


Fig. 5. QConfig Construction – Booster generates QConfig objects from interesting configurations (e.g., configurations that improved the user's objective function): C0, C2, and C5. As Q1-C5 illustrates, each QConfig contains information (e.g., knobs, plan) about a specific query in a configuration, a link to a downstream configuration (e.g., Q1-C2 to Q1-C5), and multiple identity vectors (i.e., embeddings) obtained by passing different schematics through an embedder.

and root cause analysis [66]. These embedders map input texts to fixed-length vectors where related texts are located nearby [19]. By utilizing this, Booster obtains a QConfig's identity vectors by constructing representative texts (i.e., *schematics*) and passing them through an embedder.

As shown in Fig. 5, Booster builds multiple schematics from each QConfig. These schematics capture a query's semantics, ranging from high-level information about what data the query accesses to low-level details of how the DBMS will execute it. Each schematic comprises two parts: (1) schema metadata and (2) the query. The schema metadata describes the referenced relations and columns, utilized indexes, and additional statistics (e.g., number of relation tuples, number of distinct values for each column). The query component describes what the DBMS executes, which can take the form of the SQL query, the query template, or query plan. Booster generates schematics based on all three, along with variants derived by anonymizing table and column names. It then passes each schematic through an embedder to obtain an identity vector (i.e., embedding) and loads the assembled QConfig into a vector database.

4.2 QConfig Guided Recommendation

With the vector database from Phase I, Booster in Phase II obtains candidate configurations with a LLM following the process in Fig. 6. For a given user query, Booster generates schematics and computes identity vectors (IVs) using the same process and embedder as in Phase I. With each query's IVs, Booster retrieves similar QConfigs by minimizing the euclidean distance to each QConfig's IV of the same schematic type. For example, assume Booster uses the anonymized query plan to build schematic S and obtains IV1. When searching the vector store with IV1, Booster only considers each QConfig's IV that is built with schematic S. This ensures that Booster does not erroneously rank QConfigs based on differences in schematic type.

However, the most semantically similar QConfig may not be the best reference. Consider the case where the DBMS is in the stock configuration and Booster is analyzing a previously tuned query. In this case, the most semantically similar historical QConfig′ is derived from the same query in the same stock configuration. Yet, this QConfig′ lacks any guidance (e.g., query knobs to set, indexes to build) to extract. Instead, a more relevant QConfig* with guidance exists downstream of QConfig′ by following the links. Based on this, we make a core change to retrieval: from each retrieved QConfig, Booster follows its links to fetch the most performant downstream QConfig.

Due to LLM context window limitations (i.e., prompt and response length) [36, 54], Booster truncates the references to the top-k QConfig, packages the reference QConfigs into the prompt,

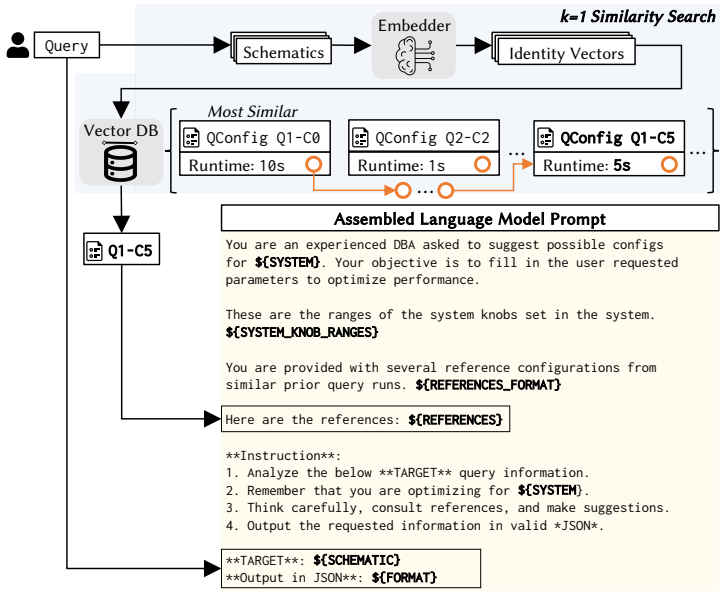


Fig. 6. Prompt Augmentation with k=1 Relevant QConfig – Based on the query, Booster derives identity vectors in a similar process to Fig. 5. It then retrieves a ranked list of QConfigs based on similarity (i.e., Euclidean distance), takes the most similar QConfig, and follows its links to obtain the most performant downstream QConfig. Booster then enriches the prompt with the downstream QConfig (QConfigQ1-C5) and prompts the LLM for suggested configurations.

and instructs the LLM to reason through those references (Instruction #3 in Fig. 6) [39, 47] to suggest configurations. Based on empirical trials, Booster sets k to a default value of 2.

4.3 Recommendation Sanitization

Prior techniques [29, 35] use the configurations suggested by the LLM "as-is". Booster forces the LLM to output valid JSON to avoid the complexity of extracting relevant configuration snippets. However, these suggestions may be incomplete, contain invalid indexes and parameters (e.g., a knob from a different DBMS), or request to match parameter values to the references. To correct these errors, Booster's Sanitizer cleans the suggested configurations using each query's QConfig references. Sanitizer first constructs a preliminary list from three sources: (1) the LLM's suggestions, (2) the QConfig references, and (3) additional configurations obtained by filling in any missing parameters in the LLM's suggestions with values from QConfig references.

As this set may occupy a locally suboptimal region, Sanitizer introduces a limited degree of exploration. From each preliminary configuration, Sanitizer derives additional candidates in two ways to produce diverse query plans: (1) augment indexes based on static analysis of the query's predicates and (2) permuting the query knobs (e.g., turn off sorting). For example, turning off sorting in PostgreSQL while keeping other query knobs (e.g., access method hints) the same tends to produce different yet performant plans. We provide a further sensitivity analysis in Sec. 8.2.

Sanitizer then removes invalid selections (e.g., unknown columns, invalid hints) from each candidate configuration before obtaining its minimal form. For example, Sanitizer uses a "what-if" mechanism [17] to identify indexes used by each candidate and then removes the unused indexes. Another example is query hints that allow multiple possibilities. For instance, PostgreSQL's `NoSeqScan(t)` [4] hint requests the DBMS to use an index scan if possible. Sanitizer modifies these

Algorithm 1 Beam Search

Input: Query-Configs $QC = \{\ldots, (q_i, [c_{i1}, \ldots, c_{ij}])\}$
Output: Target Configuration
$seeds = \{(q_i, best(c_i)) \forall (q_i, c_i) \in QC\}$ ▷ 1
$cands = \texttt{Merge}(seeds)$ ▷ 2
$best = \texttt{Evaluate}(cands)$ ▷ 3
while budget not exhausted **do**
  $q_{next} = \texttt{Select}(best)$ ▷ 4
  $alt_q_seeds = \texttt{Rollout}(q_{next}, best, QC)$ ▷ 5
  $cands = \texttt{Merge}(best, q_{next}, alt_q_seeds)$ ▷ 6
  $best = \texttt{Evaluate}(cands)$
end while
Return *best*

hints based on the query plan to their more specific form (e.g, `IndexScan(t)`). Finally, it preserves all configurations that result in unique query plans as the query's seeds.

5 Constrained Composition

Once Booster obtains unique seeds for each query in Phase II (see Sec. 4.2), it combines them into a holistic configuration through a *rank*-and-*compose* process. We discuss each part separately.

5.1 Ranking Seeds

As a query's seeds can have diverse performance outcomes, ranking them is crucial for composing into a holistic configuration. The simplest approach is to use the DBMS's estimated plan cost. However, plan cost is not necessarily correlated with plan quality [13], particularly when query hints are involved that distort the cost. Alternatively, there is extensive literature in learned cost models [33, 57, 59] that build models to predict plan runtime. However, they require substantial representative training data and are shown to have limited efficacy for ranking query plans [32].

Booster instead executes each seed to estimate its quality. Deploying all seeds (e.g., building all indexes) induces significant overhead. Thus, Booster derives alternate indexes that cover the original indexes [77]. To estimate each seed, Booster executes it after replacing the original plan's indexes with alternate indexes that upper bound the original plan's runtime. For example, consider three seeds: (1) Q1 with I1 `t(a,b)`, (2) Q2 with I2 `t(a)` and Q2 does not access `b`, and (3) Q3 with I3 `t(a)` and Q3 does access `b`. For Q1 and Q2, Booster builds I1 and forces both to use it. However, if Booster forces Q3 to use I1, then Q3 may avoid heap fetches with the covering index I1 and execute faster than otherwise possible. To avoid this, Booster builds I3 and forces Q3 to use I3.

Booster executes all seeds in order of increasing plan cost. If the DBMS distorts the cost due to query hints (e.g., turn off sorting when the query requires sorting), Booster computes a hypothetical cost without distortions. To control query execution overhead, Booster adopts a simple per-query timeout strategy. Although Booster could prune seeds with rules or based on clustering analysis of query plans (e.g., common access paths, similar partial join order), there are no established strategies to do so, particularly when queries time out and do not return any execution information (e.g., EXPLAIN ANALYZE). We defer a principled investigation of pruning to future work.

5.2 Composition Algorithm

After ranking seeds, Booster composes a holistic configuration. Inspired by relaxation-based physical design [15], Booster greedily constructs an initial configuration and then uses beam search (i.e., best-first search [12, 20, 69]) to refine it. In the best case, combining each query's best seed results in a near-optimal configuration. If conflicts arise (e.g., a query's runtime degrades from its seed's runtime), Booster alters the configuration by targeting those conflicting seeds.

Algorithm 1 shows the beam search algorithm that runs until it exhausts the time budget. ❶ Booster obtains each workload query's best seed, ❷ composes them into holistic candidates (**Merge**), ❸ evaluates them, and selects the best one. From the best candidate, ❹ Booster selects a query seed to refine (**Select**), ❺ rolls out alternate seeds for the selected seed (**Rollout**), and ❻ repeats from ❷ by replacing the selected query seed with alternate ones. Our evaluation in Sec. 8.3 shows that composing for 1.5 hours produces good results. We next discuss these three steps.

Merge: In this step, Booster composes the seed for each query into a holistic configuration. As these seeds may have incompatible system knobs (e.g., different parallel workers) or physical design structures, Booster must reconcile them. For system knobs, Booster generates configurations based on the minimum, median, and maximum across the seeds. For physical design structures, Booster takes the union and removes identical ones. It then runs these holistic configurations on the DBMS with a cache to eliminate identical plan invocations [52].

Select: In this step, Booster picks the next query to refine. Booster first fixes conflicts (i.e., degraded queries) from merging into a holistic configuration. During this phase, Booster greedily picks a query that performed worse than estimated. Afterwards, Booster greedily picks the query with the highest runtime to refine. Booster tracks selected queries in query-agnostic configurations (i.e., only system knobs and indexes) to avoid re-selecting Q1 if Q2 only undergoes a local change (e.g., query hints) that do not impact Q1.

Rollout: In this step, Booster generates alternate seeds for the selected query. It then merges each alternate seed with the other queries' seeds into holistic configurations and evaluates them. Booster generates these alternate seeds with the following constrained exploration mechanisms (**M1–4**). We analyze these further in Sec. 8.5.

- **M1 Query Knob Permutation**: Similar to in Phase II (see Sec. 4.3), this mechanism permutes the seed's query knobs to generate diverse plans (i.e., seeds). Common variations include deferring to the optimizer [21, 96] or forcing a nested loop join [46, 58].
- **M2 Plan Repair**: This corrects the query seed's plan under the holistic configuration to match its plan in isolation. Booster analyzes both plans and then generates alternate seeds by modifying query hints to restore or eliminate nodes. For example, if the holistic configuration's plan contains a `Sort` node that is not present in isolation, Booster creates an alternate seed with sorting disabled. Booster attempts three repairs: (1) enable optimizer flags corresponding to used nodes, (2) turn off nodes not in the isolated seed's plan, and (3) enable nodes in the isolated seed's plan.
- **M3 Hidden Indexes**: Prior work notes that combining index sets together may impair cost-based query optimizers [15]. This mechanism aims to uncover alternative performant plans. After obtaining the indexes used by the query seed, Booster builds alternate seeds that omit them.
- **M4 Ranked Seeds**: Each query has many ranked seeds that may compose with the other queries' seeds into different holistic configurations. To improve the query, Booster only selects ranked seeds with estimated runtimes less than the present query seed's runtime.

Due to the above mechanisms, Booster may generate and evaluate a large number (e.g., ≈100) of holistic configurations. To account for this, Booster adopts a 5-step rollout: (1) **Local**, (2) **25% Seed**, (3) **50% Seed**, (4) **75% Seed**, and (5) **100% Seed**. In **Local**, Booster uses **M1,M2** to try query-local alternative seeds that are fast to evaluate with per-query timeouts to bound the runtime. In the other steps, Booster explores seeds from **M3,M4** in slices based on estimated runtime with a workload timeout. Thus, Booster avoids suboptimal seeds if it discovers a performant one in an earlier step.

6 Integration

We now discuss how Booster integrates with existing tuners. As Booster is tuner-agnostic, developers can plug it into a tuner through its API commands: (1) `Parse`, (2) `Link`, and (3) `Digest`.

`Parse`. As tuners generate artifacts in various formats, this command enables Booster to decipher them and extract their key insights. This process reconciles differences in tunables (e.g., knobs, query hints) between what a tuner supports and what Booster reasons over. For example, some tuners only consider whether sequential scans or index scans are enabled system-wide [86, 98] or on a per-query [11, 58] basis. By contrast, Booster reasons over access methods at a finer table-level granularity for each query.

`Link`. Developers implement how Booster links QConfigs. Recall that the QConfig retrieval process uses these links to retrieve semantically relevant and performant QConfigs (see Sec. 4.2). We propose linking based on the temporal nature of tuning steps. By default, Booster links QConfig objects based on explored trajectories: C1 links to C2 if C2 is reached (i.e., explored) from C1.

`Digest`. Lastly, once Booster completes its composition, Booster passes its findings to the tuner for further refinement. By default, Booster exposes its best holistic configuration to the tuner through `Digest`. However, Booster could expose additional artifacts (e.g., explored configurations) or guide further tuning focus (e.g., target specific queries). We envision that future tuners will support the ingestion of this data. We defer this to future work.

We next discuss how we envision operators will deploy Booster. An operator must provide Booster with an offline environment [55] and API access (or local GPU) to its embedder and prompter LLMs. As Booster and, by extension, downstream tuners are unable to estimate the benefit of further tuning, we rely on the operator to judge when to invoke Booster. Upon starting a new tuning session, we expect that the operator will first invoke Booster, then run any downstream tuners, and finally ensure that Booster analyzes any new artifacts. We defer mechanisms that balance cost and the benefit of further tuning to future work.

7 Evaluation

We evaluate Booster's ability to accelerate the convergence of existing tuners when confronted by different drifts and transfer scenarios. We target PostgreSQL v15.1 running on a server with two Intel Xeon Gold 5218R CPUs (20 cores) and a 960 GB Samsung NVMe SSD. We restrict the DBMS to 32 GB of RAM and 20 worker processes. To support Booster's operations in PostgreSQL, we install *HypoPG* [3] for what-if mechanism [17], *pg_hint_plan* [4] for query tuning, and a custom *HypoExec* extension for re-costing query plans and swapping indexes during execution.

We primarily evaluate with three OLAP workloads. **JOB** [46] (6.7 GB) is a benchmark that stresses the query optimizer with 21 tables and 113 queries. **TPC-H SF10** [2] (14.3 GB) models a business analytics workload with eight tables and 22 queries. **DSB SF10** [23] (22.5 GB) is Microsoft's extension of TPC-DS [81] that introduces additional challenges (e.g., data distributions, join patterns) with 25 tables and 53 queries. We omit four queries (Q18, Q32, Q81, Q92) due to PostgreSQL's query optimizer's limited ability to unnest subqueries [27].

7.1 Experiment Setup

We deploy four state-of-the-art DBMS tuners:

PGTune+DTA+AutoSteer (P+DTA+AS). This first runs PGTune [1], a heuristics-based knob tuner, followed by Microsoft's Anytime Database Tuning Advisor (DTA) algorithm [18]. We use Hyrise's

Table 1. Phase I: Experience Analysis Overhead – We present the mean # QConfigs, # tokens, and upper bound the embedder cost. Booster invokes the Voyage 3 Large API at $0.18/million tokens with a rate limit of 3 million tokens/min [7]. In practice, Booster caches the API call's inputs and outputs to reduce cost.

	# QConfigs	# Million Tokens	Embedder Cost (Upper Bound)
DSB (49 Queries)			
Proto-X	1852	17.8	$3.2
P+DTA+AS	3451	38	$6.8
UniTune	185	1.7	$0.3
λ-Tune+AS	1295	13.6	$2.4
TPC-H (22 Queries)			
Proto-X	373	1.6	$0.3
P+DTA+AS	4662	5.9	$1.1
UniTune	94	0.4	$0.1
λ-Tune+AS	592	2.9	$0.5
JOB (113 Queries)			
Proto-X	3325	25.2	$4.5
P+DTA+AS	4628	44.5	$8
UniTune	791	6.1	$1.1
λ-Tune+AS	1732	15.5	$2.8

implementation of DTA [40] with an unlimited tuning budget. After DTA finishes, we run AutoSteer [11] to tune query knobs by greedily toggling and merging boolean knobs.

UniTune. Alibaba's coordinating tuning framework that targets system knobs, indexes, and limited query rewriting through Calcite [96]. We adopt the same settings as their paper, but modify it to run queries serially and to minimize the workload runtime.

Proto-X. CMU's holistic tuner that targets system knobs, indexes, and query options supported by pg_hint_plan [96]. We adopt the same parameters from their paper and extend it to suggest index or bitmap scans and common table expressions (CTEs) inlining.

LambdaTune+AutoSteer (λ-Tune+AS). This first runs Cornell's LLM-based tuning agent (LambdaTune [29]) that analyzes the workload, constructs a tuning prompt based on the analysis, and obtains configurations from GPT-4o [63]. After obtaining a configuration, it then runs AutoSteer [11] to tune query knobs.

We next briefly discuss Booster's standard configuration for all experiments.[1] For local LLM inference, Booster uses an RTX3080 GPU with 10GB of RAM, an Intel Xeon W1350 CPU (12 cores) and 32GB of RAM. Booster uses Voyage 3 Large [6] for its embedding model to generate identity vectors (i.e., embeddings) and Llama 3.1-8B-Instruct Q4-K-M [30] to suggest configurations. For consistency, we invoke the LLM with a temperature of 0, a context window of 16384 tokens, and maximum output of 4096 tokens.

In Phase I, we configure Booster to reason over configurations that span Proto-X's options: system knobs, indexes, and query options for each query that cover optimizer switches, access method selection (e.g., sequential, index, bitmap), and whether to materialize or inline CTEs. In Phase II, Booster uses all permutation strategies to generate query seeds (Sec. 4.3). In Phase III, Booster runs its constrained composition for 1.5hr (Sec. 5.2).

We populate a tuning repository for each tuner by running four trials with random seeds on the same hardware. At the start of each trial, we initialize PostgreSQL with its default configuration. We then run each tuner for 12hr to tune the DBMS. While these tuners run, Booster in Phase I (Sec. 4.1) analyzes their artifacts and generates QConfigs that are available once the tuners finish. We present the overhead of the phase in Tab. 1.

For tuners *continuing from history*, we start each tuner from one of the best configurations in the repository and allow it 12hr to optimize the DBMS further. When accelerating a tuner with Booster,

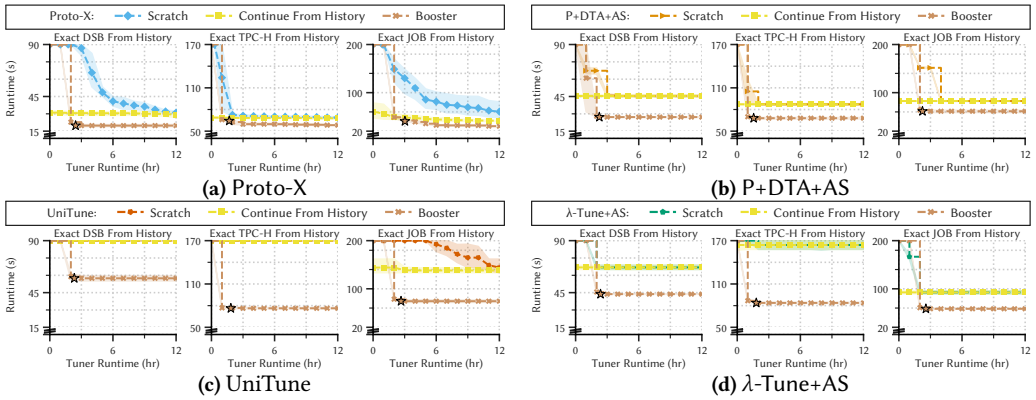
[1]Source Code: https://anonymous.4open.science/r/adamantine-867C/.

Table 2. Exact Transfer Phase II: Guided Recommendation Embedder Costs – We present the mean embedder (Voyage 3 Large) costs. These costs depend only on the workload and not the assisted tuner.

	# Queries	# Tokens	Runtime	Embedder (Voyage 3 Large) Cost
DSB	49	450k	3.4min	$0.08
TPC-H	22	92k	1min	$0.02
JOB	113	837k	5.6min	$0.15

Table 3. Exact Transfer Phase II: Guided Recommendation Prompter Costs – We present the mean prompter (Llama 3.1-8B-Instruct Q4-K-M) costs across each tuner's trials. We provide estimates assuming a 320W load on our local RTX3080 and estimate cloud inference costs from OpenRouter [5].

	# Input Tokens	# Output Tokens	Runtime	Est. Power	Est. OpenRouter Cost
DSB					
Proto-X	261k	39k	9.8min	0.05kWh	$0.005
P+DTA+AS	278k	37k	9.8min	0.05kWh	$0.005
UniTune	270k	36k	9.3min	0.05kWh	$0.005
λ-Tune+AS	275k	40k	10.2min	0.05kWh	$0.01
TPC-H					
Proto-X	68k	14k	3min	0.02kWh	$0.001
P+DTA+AS	71k	12k	3min	0.02kWh	$0.001
UniTune	68k	13k	2.9min	0.02kWh	$0.001
λ-Tune+AS	70k	13k	2.9min	0.02kWh	$0.001
JOB					
Proto-X	537k	80k	20min	0.11kWh	$0.01
P+DTA+AS	510k	88k	20min	0.11kWh	$0.01
UniTune	542k	86k	21min	0.11kWh	$0.01
λ-Tune+AS	518k	96k	22min	0.12kWh	$0.01



(a) Proto-X **(b)** P+DTA+AS **(c)** UniTune **(d)** λ-Tune+AS

Fig. 7. Exact Transfer – The DBMS's performance achieved by each framework on DSB, JOB, and TPC-H when tuning from scratch, from the best historical configuration, and accelerated with Booster. We plot the mean performance obtained by four trials of each tuner, with the error band representing the 95% confidence interval. For Booster, the ⋆ illustrates when it finishes composing a holistic configuration.

each Booster invocation runs Phase II and III with access to all artifacts from that tuner's repository (e.g., four trials of Proto-X). After each tuner's trial, we re-evaluate discovered configurations without timeouts in a warm cache to obtain their actual performance [45, 96].

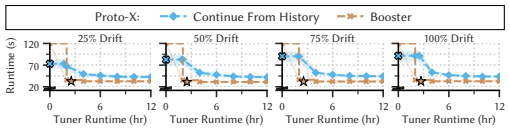
7.2 Exact Transfer

We first evaluate Booster's ability to accelerate tuners when tuning the same deployment from history. We first present the embedder and prompter costs incurred by accelerating with Booster in Tabs. 2 and 3. We then compare three tuner variations: (1) tuning from scratch, (2) continuing from the best historical configuration, and (3) accelerated with Booster. We report each tuner's best configuration mean performance, the 95% confidence interval band, and the ⋆ to indicate when Booster completes. We also show each tuner's worst, mean, and best performance in Tab. 4.

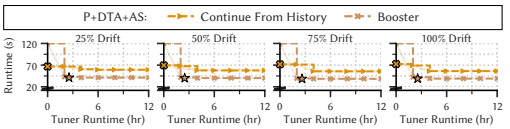
As shown in Fig. 7, **Continue From History** is generally not an effective strategy. P+DTA+AS and λ-Tune+AS (Figs. 7b and 7d) exhibit no improvement, whereas UniTune (Fig. 7c) achieves only

Table 4. Exact Transfer Performance Spread – The worst, mean, and best result for the tuners' trials in Fig. 7 on DSB, TPC-H, and JOB when tuning from scratch, continuing from history, and with Booster.

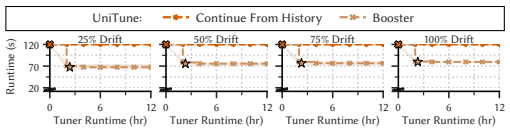
	Scratch			Continue			Booster		
Benchmark	Min	Mean	Max	Min	Mean	Max	Min	Mean	Max
DSB→DSB									
Proto-X	29s	31s	33s	26s	29s	32s	**19s**	**20s**	**21s**
P+DTA+AS	44s	45s	47s	44s	45s	47s	**27s**	**27s**	**27s**
UniTune	125s	136s	148s	125s	136s	148s	**55s**	**57s**	**60s**
λ-Tune+AS	64s	67s	70s	64s	67s	70s	**44s**	**44s**	**44s**
TPC-H→TPC-H									
Proto-X	68s	69s	71s	66s	68s	69s	**58s**	**58s**	**59s**
P+DTA+AS	86s	87s	88s	86s	87s	88s	**66s**	**68s**	**69s**
UniTune	199s	200s	201s	176s	194s	200s	**76s**	**76s**	**78s**
λ-Tune+AS	156s	167s	181s	156s	167s	181s	**83s**	**84s**	**85s**
JOB→JOB									
Proto-X	47s	61s	88s	40s	41s	42s	**29s**	**30s**	**32s**
P+DTA+AS	82s	83s	83s	82s	83s	83s	**57s**	**61s**	**64s**
UniTune	132s	144s	170s	132s	138s	146s	**72s**	**74s**	**78s**
λ-Tune+AS	89s	93s	97s	89s	93s	97s	**58s**	**58s**	**60s**



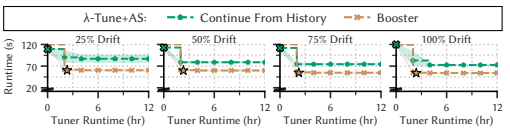
(a) Proto-X



(b) P+DTA+AS



(c) UniTune



(d) λ-Tune+AS

Fig. 8. Parameter Drift – The DBMS's performance achieved by each framework in response to workload parameter drift. We plot the mean performance with a 95% confidence interval obtained by four trials of each tuner. The **X** represents the starting point when continuing from history. For Booster, the ★ illustrates when it finishes composing a holistic configuration.

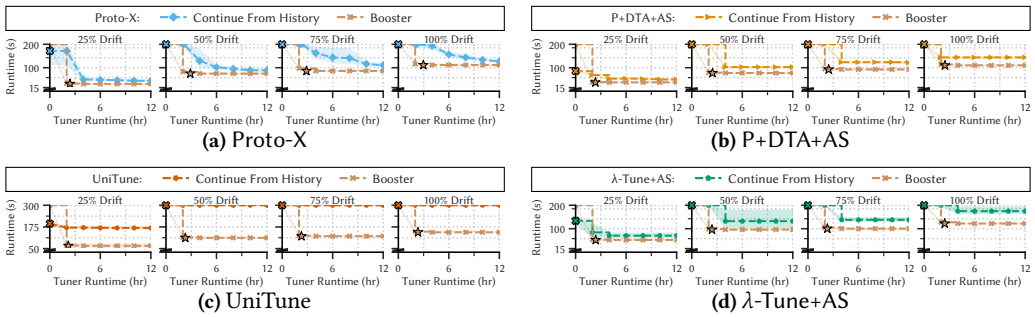
3–4% mean improvement on TPC-H and JOB. Tab. 4 indicates that although Proto-X achieves only 1% and 6% improvement on TPC-H and DSB, respectively, its JOB configuration is 33% better.

These results show Booster enables tuners to discover configurations that are better than tuning from scratch or continuing from history. Booster helps to find configurations that have a mean improvement of 16–51% (Proto-X), 22–40% (DTA), 49–62% (UniTune), and 34–50% (λ-Tune+AS) over tuning from scratch. Booster achieves this by analyzing artifacts for query-level insights and composing those insights together with constrained exploration. It extends the set of tunables (e.g., query hints) supported by the tuners and also breaks their search algorithms out of local optima.

7.3 Parameter Drift

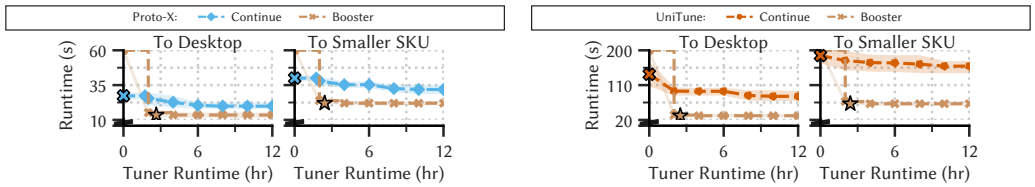
We next evaluate Booster's ability when the workload undergoes a parameter drift, whereby only the parameters of the queries change. We generate a set of DSB queries with a different seed and replace portions of the historical workload: **25%**, **50%**, **75%**, and **100%**. We run four 12hr trials for each drift percentage and plot the mean performance along with 95% confidence interval in Fig. 8.

As shown in Fig. 8 with the ★ marker, Booster consistently outputs a holistic configuration in under 3hr. With Booster, all frameworks find configurations with mean improvements of 23–27% (Proto-X), 31% (P+DTA+AS), 42–64% (UniTune), and 24–31% (λ-Tune+AS) over those found by continuing to tune from historical configurations. Booster re-mixes query seeds based on query semantics, rather than the entire workload. For repeated queries, Booster extracts beneficial seeds



(a) Proto-X (b) P+DTA+AS (c) UniTune (d) λ-Tune+AS

Fig. 9. Template Drift – The DBMS's performance achieved by each framework in response to workload template drift. We plot the mean performance with a 95% confidence interval obtained by four trials of each tuner. The **X** represents the starting point when continuing from history. For Booster, the ★ illustrates when it finishes composing a holistic configuration.



(a) Proto-X (b) UniTune

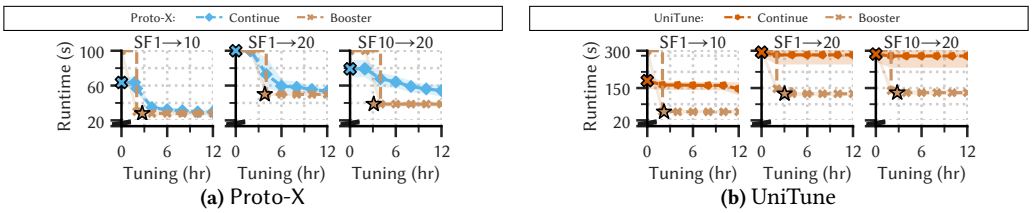
Fig. 10. Machine Transfer – The DBMS's performance achieved by each framework in response to machine transfer. We plot the mean performance with 95% confidence interval obtained by four trials of each tuner. The **X** represents the starting point when continuing from history. For Booster, the ★ illustrates when it finishes composing a holistic configuration.

"as-is". For queries with changed parameters, Booster generalizes from historical seeds with the same query template. Thus, it exploits the observation that queries with the same templates may benefit from similar optimizations. Booster assists in finding these configurations up to 3.6× (Proto-X), 1.9× (DTA), and 3.6× (UniTune) faster. λ-Tune+AS finds configurations 10–30min faster than when using Booster. Booster spends ~10min longer on LLM inference, as it prompts the LLM on a query rather than workload granularity. Nevertheless, λ-Tune+AS with Booster still finds configurations that are 24–31% better than without.

7.4 Template Drift

We now assess how Booster manages workloads that experience template drifts. We generate TPC-DS [81] queries that are not also DSB templates and replace portions of the historical workload by **25%**, **50%**, **75%**, and **100%**. We run four 12hr trials per percentage and plot the mean performance along with 95% confidence interval.

Similar to the previous experiment, Fig. 9 shows that Booster outputs a holistic configuration in under three hours (★ marker). Booster enables the tuners to find configurations with mean improvements of 12–30% (Proto-X), 24% (P+DTA+AS), 52–63% (UniTune), and 26–39% (λ-Tune+AS) over those found by tuning from historical configurations. Booster assists in finding these configurations up to 4.7× (Proto-X), 3.4× (P+DTA+AS), 2.7× (UniTune), and 1.2× (λ-Tune+AS) faster. Booster handles unseen queries through two strategies. It first generalizes from similar historical queries. In cases where the LLM generates a locally suboptimal configuration, Booster uses its constrained exploration mechanism to derive similar, more performant configurations (Sec. 4.3).



(a) Proto-X (b) UniTune

Fig. 11. Dataset Growth – The DBMS's performance achieved by each framework in response to dataset growth. We plot the mean performance with a 95% confidence interval obtained by four trials of each tuner. The **X** represents the starting point when continuing from history. For Booster, the ★ illustrates when it finishes composing a holistic configuration.

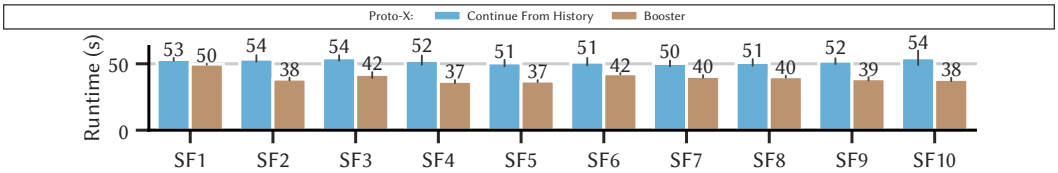


Fig. 12. Dataset Growth Sensitivity – Mean performance with 95% confidence interval obtained by four trials of Proto-X. Evaluates the dataset growth scenario from different DSB SF to SF20.

7.5 Machine Transfer

We next evaluate Booster's ability when the environment exhibits a hardware change. We evaluate two scenarios moving from a high-performance server to a *weaker* (Intel Xeon Silver 4114) and *desktop* (Intel Xeon W-1350) machine with the same historical DSB workload. We evaluate the best- and worst-performing tuners from Sec. 7.2 using four 12hr trials for each environment.

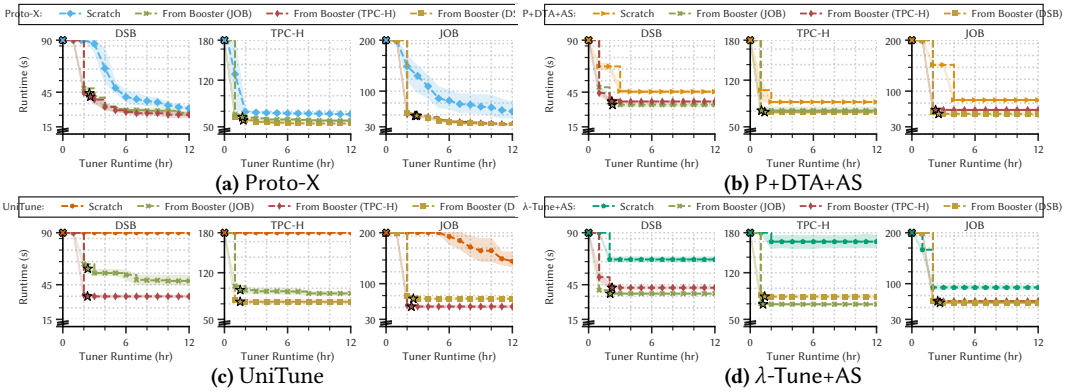
As shown in Fig. 10, all tuners with Booster find configurations with mean improvements of 31–32% (Proto-X) and 61–62% (UniTune) over those found by tuning from historical configurations. Furthermore, Booster enables tuners to find these configurations up to 3.1× (Proto-X) and 2.2× (UniTune) faster. As the target workload is the same as the history, Booster recognizes that each query's best historical seed is a promising starting point. It then leverages this to assist both tuners in finding better configurations more quickly.

7.6 Dataset Growth

Another challenge for tuners is when the database grows over time or when switching from development to production instances. To measure this effect, we generate three DSB variations: **SF1→10**, **SF1→20**, and **SF10→20**. Following the process in Sec. 7.1, we first construct the artifacts repository from tuning **SF1**. We then evaluate the best- and worst-performing frameworks from Sec. 7.2 using the same workload for four 12hr trials per scenario.

Fig. 11's ★ marker indicates that Booster takes longer on **SF20** due to the higher query execution overhead. As shown in Fig. 11b, Booster assists UniTune in finding configurations with mean improvements of 63% (**SF1→10**), 55% (**SF1→20**), and 53% (**SF10→20**) over those found by tuning from historical configurations.

In contrast in Fig. 11a, Proto-X shows more modest improvements of 6% (**SF1→10**), 7% (**SF1→20**), and 29% (**SF10→20**). To investigate the performance difference between adapting to **SF20** from **SF1** versus **SF10**, we sweep each SF from 1 to 10 and evaluate with the same methodology as above. We run four 12hr trials for each scenario and plot the mean performance along with 95% confidence interval in Fig. 12. As Fig. 12 shows, adapting from SF1 stands out, with the others resulting in configurations that are 18–30% better.



(a) Proto-X (b) P+DTA+AS

(c) UniTune (d) λ-Tune+AS

Fig. 13. Cross-Schema Transfer – The DBMS's performance achieved by each framework on DSB, JOB, and TPC-H when tuning from scratch and accelerated with Booster from other benchmarks' artifacts. We plot the mean performance obtained by four trials of each tuner, with the error band representing the 95% confidence interval. For Booster, the ★ illustrates when it finishes composing a holistic configuration.

Table 5. Cross-Schema Transfer Performance Spread – The worst, mean, and best performance achieved by a framework's four trials in Fig. 13 on DSB, TPC-H, and JOB when tuning from scratch, and with Booster from other benchmarks' repository artifacts.

	Scratch			Booster (DSB)			Booster (TPC-H)			Booster (JOB)		
Benchmark	Min	Mean	Max	Min	Mean	Max	Min	Mean	Max	Min	Mean	Max
DSB												
Proto-X	29s	31s	33s	-	-	-	22s	**26s**	28s	25s	**26s**	27s
P+DTA+AS	44s	45s	47s	-	-	-	36s	37s	37s	32s	**34s**	36s
UniTune	125s	136s	148s	-	-	-	34s	**35s**	36s	44s	48s	55s
λ-Tune+AS	64s	67s	70s	-	-	-	41s	42s	44s	36s	**37s**	38s
TPC-H												
Proto-X	68s	69s	71s	54s	**55s**	56s	-	-	-	58s	59s	61s
P+DTA+AS	86s	87s	88s	70s	**72s**	74s	-	-	-	71s	74s	82s
UniTune	199s	200s	201s	74s	**76s**	80s	-	-	-	85s	89s	92s
λ-Tune+AS	156s	167s	181s	82s	84s	88s	-	-	-	72s	**73s**	73s
JOB												
Proto-X	47s	61s	88s	30s	**34s**	37s	34s	35s	35s	-	-	-
P+DTA+AS	82s	83s	83s	54s	**55s**	57s	62s	63s	65s	-	-	-
UniTune	132s	144s	170s	69s	71s	72s	53s	**55s**	58s	-	-	-
λ-Tune+AS	89s	93s	97s	61s	**63s**	64s	64s	66s	67s	-	-	-

Upon further inspection, we find that a single query accounts for ≈7s of the 12s difference between SF1→SF20 and SF2–10 → SF20. The issue begins when Proto-X overfits to SF1 and selects holistic configurations (e.g., turn off hash aggregates, ignore covering indexes) that improve performance by milliseconds without considering stability or generalizability. When adapting to SF20, Booster is trapped by those SF1 configurations in local optima and cannot break out without an expensive combinatorial search. In practice, changes to plans or plan performance happen for various reasons, such as software upgrades or dataset growth. As Booster effectively reuses historical artifacts to accelerate re-tuning, we expect operators to run Booster more frequently to correct those changes. We defer building tuners with an explicit stability or generalizability objective, in addition to a performance objective, for future work.

7.7 Cross-Schema Transfer

We next evaluate Booster's ability to accelerate tuners when tuning a new DBMS using artifacts from a different application's database. We run four trials for each tuner variation across all workloads. We again plot the mean performance of each tuner's best configuration, the 95% confidence interval band, and use ★ to indicate when Booster completes. We also report the worst, mean, and best performance achieved by any framework's trial in Tab. 5.

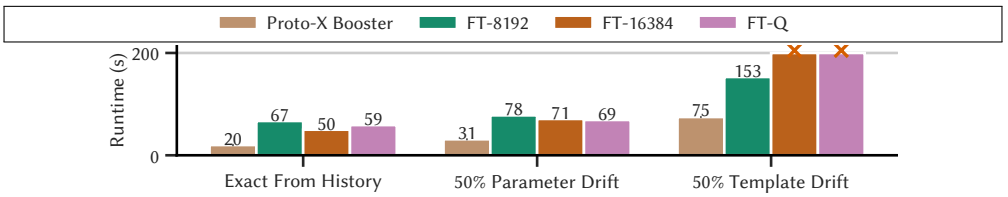


Fig. 14. Fine-Tuning – The DBMS's performance achieved by each technique across three scenarios based on Proto-X's historical DSB tuning artifacts. We plot the mean performance with a 95% confidence interval obtained from four trials of each technique *without* further refinement.

As indicated in Fig. 13, tuning from the best historical configuration is identical to tuning from scratch. Due to schema differences, existing tuning frameworks cannot transfer the historical configurations. By contrast, Booster obtains holistic configurations for the target DBMS that are inspired by artifacts, with configurations that leverage `INCLUDE` columns (i.e., covering index) and index knobs (e.g., page fillfactor). Thus, Booster assists tuners in discovering configurations with a mean improvement of 14–44% (Proto-X), 15–33% (P+DTA+AS), 51–74% (UniTune), and 29–56% (λ-Tune+AS) over starting with the best historical configuration. Tab. 5 shows that there is no dominant benchmark (i.e., artifact repository) to initialize Booster with. Instead, Booster can use diverse artifacts to accelerate tuners in finding better configurations.

8 Sensitivity Experiments

We next analyze aspects of Booster in more detail. We begin with an ablation study on fine-tuning in Sec. 8.1, query knob permutation strategy (Sec. 4.3) in Sec. 8.2, the composition phase's search time in Sec. 8.3, the embedder and prompter LLMs in Sec. 8.4, the composition phase's rollout policy (Sec. 5.2) in Sec. 8.5, and size of the input data in Sec. 8.6.

8.1 Fine-Tuning

An alternative to Booster's approach of enriching the prompt with historical references is fine-tuning an LLM on (prompt, configuration) pairs [35]. We study whether fine-tuning alone is sufficient. Using the most performant tuner Proto-X's DSB artifacts in the transfer, parameter, and template experiments (Secs. 7.2 to 7.4), we extract each trial's steps into workload-level training data pairs. For each step, we set the output to the trial's best configuration and construct a prompt that contains tuning instructions, a workload summary, DBMS metrics, and query plans truncated to specific context lengths (i.e., **FT-8192**, **FT-16384**). We explore a query variant (**FT-Q**) that replaces the workload summary with the SQL query.

We fine-tune Booster's LLM (Llama 3.1-8B-Instruct) with LLaMa-Factory [103], FlashAttention-2 [22], and DeepSpeed [71]. For all, we use a learning rate of 2e-5 for 8 epochs [35]. We use 4 RTX3090 for **FT-8192** and **FT-Q** and 2 RTXA6000 for **FT-16384**. During inference for **FT-8192** and **FT-16384**, we sample 8 configurations at a temperature of 0.2 [35] and select the best. For **FT-Q**, we sample three configurations for each query with a temperature of 0.2, select each query's best config, and then combine them together.

We consider three scenarios from the same historical DSB workload: **Exact From History**, **50% Parameter Drift**, and **50% Template Drift**. We run four trials for each technique *without* further refinement. We plot the mean performance along with a 95% confidence interval in Fig. 14. Across all scenarios, fine-tuning performs worse. For **FT-8192** and **FT-16384**, truncating to context lengths restricts learning to the prompt's queries. For **FT-Q**, query configs conflict when combined, as seen in **50% Template Drift**.

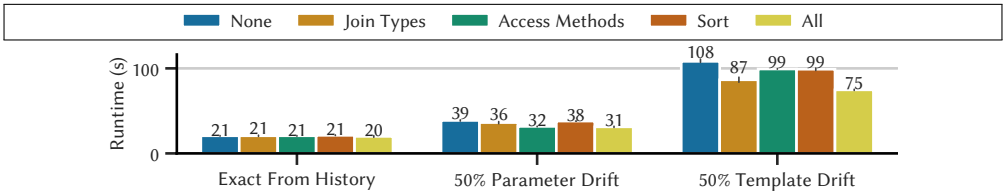


Fig. 15. Query Knob Permutation Strategy – The DBMS's performance achieved across three scenarios based on Proto-X's historical DSB tuning artifacts when varying Booster's query knob permutation strategy. We plot the mean performance with a 95% confidence interval obtained from four trials of each technique *without* further refinement.

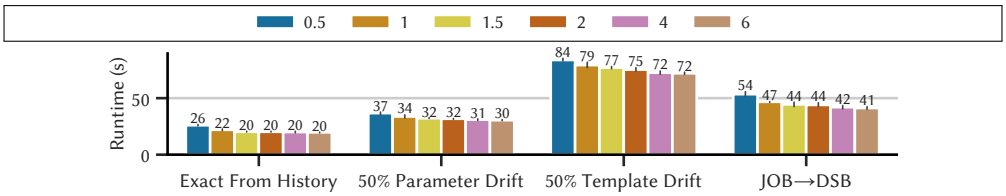


Fig. 16. Search Time – The DBMS's performance achieved across four scenarios based on Proto-X's historical DSB tuning artifacts when varying Booster's search time. We plot the mean performance with a 95% confidence interval obtained from four trials of each variant *without* further refinement.

8.2 Query Knob Permutation Strategy

Booster permutes query knobs to generate similar query seeds (Sec. 4.3). We consider five strategies: **None**, **Join Types** (i.e., hash, merge, nested loop), **Access Methods** (i.e., seq-, bitmap-, or index-scan per-table), **Sort** turns off sorting, and **All**. We limit the study to the most performant tuner Proto-X and evaluate three scenarios from the same historical DSB workload: **Exact From History**, **50% Parameter Drift**, and **50% Template Drift**. We run four trials for each strategy *without* further refinement by Proto-X. We plot the mean performance along with a 95% confidence interval.

As shown in Fig. 15, the strategy does not matter for **Exact From History**, as Booster exploits the 1:1 mapping from the target workload to historical query seeds. For drifts, **Join Types**, **Access Methods**, and **Sort** strategies enable Booster to improve over **None** by allowing it to locally explore during composition to fix conflicts (i.e., degraded queries) and break out of suboptimal solutions. Booster with the **All** strategy finds the best holistic configuration, as it uses all opportunities to derive diverse plans from each query seed.

8.3 Search Time

We next vary the amount of search time allocated to Booster's constrained composition algorithm (Sec. 5.2): 0.5hr, 1hr, 1.5hrs, 2hrs, 4hrs, and 6hrs. We limit the study to the most performant tuner Proto-X and evaluate four scenarios from the same historical DSB workload: **Exact From History**, **50% Parameter Drift**, **50% Template Drift**, and **JOB→DSB**. We run four trials for each variation *without* further refinement by Proto-X. We report the mean performance along with a 95% confidence interval.

Fig. 16 shows that the trend across all four scenarios aligns with common tuning trends [82, 96]. Providing Booster more time allows it to provide better holistic configurations to the assisted tuner, with diminishing returns. Although **Exact From History** stabilizes after 1.5hrs, the other scenarios continue to show marginal improvement.

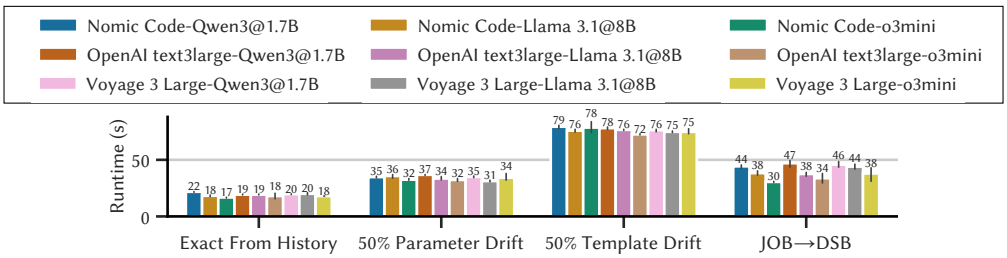


Fig. 17. Embedder-Prompter Models – The DBMS's performance achieved across four scenarios based on Proto-X's historical DSB tuning artifacts when varying the embedder and prompter LLM. We plot the mean performance with a 95% confidence interval obtained from four trials of each embedder-prompter variant *without* further refinement.

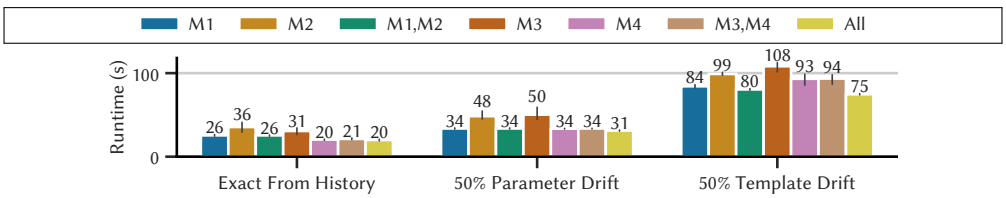


Fig. 18. Rollout Policy – The DBMS's performance achieved across three scenarios based on Proto-X's historical DSB tuning artifacts when varying Booster's rollout policy during composition. We plot the mean performance with a 95% confidence interval obtained from four trials of each variant *without* further refinement.

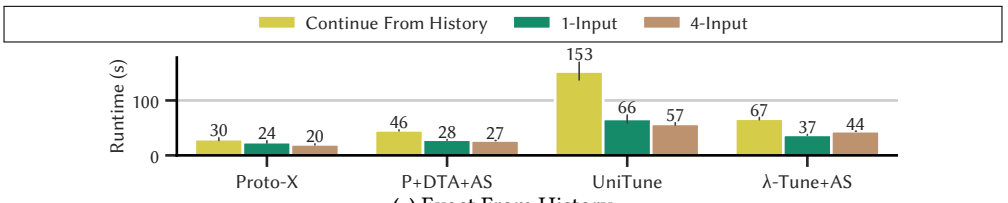
8.4 Embedder-Prompter Models

We next investigate the embedder and prompter LLMs used by Booster. We select three embedders based on the Massive Text Embedding Benchmark [61]: Nomic Code [78], text-3-large [64], and Voyage 3 Large [6]. We then select three prompters of different scales: Qwen3-1.7B [92], Llama 3.1-8B-Instruct [30], and o3-mini [65]. We limit the study to the most performant tuner Proto-X and evaluate four scenarios from the same historical DSB workload: **Exact From History**, **50% Parameter Drift**, **50% Template Drift**, and **JOB→DSB**. We run four trials for each pair *without* further refinement by Proto-X.

Examining Fig. 17, we first notice that the embedder-prompter matters more when the target workload differs from historical workloads. All pairs are comparable (≈5s) for **Exact** but have larger differences for **JOB→DSB** (≈17s). Second, increasing the complexity of the prompter LLM (i.e., # of parameters) enables Booster to find better configurations. For instance, Booster with o3-mini finds configurations that are 32% better than using Qwen3-1.7B for **JOB→DSB**. However, these improvements come with higher inference costs that are not necessary for some scenarios (**Exact**, **50% Parameter Drift**). We leave the selection of the optimal LLM based on generalizability (e.g., cross-schema) and cost requirements for future work [70].

8.5 Rollout Policy

We next study Booster's rollout policy during composition (Sec. 5.2) and consider the following variations: **M1 Query Knob Permutation**, **M2 Plan Repair**, **M3 Hidden Indexes**, **M4 Ranked Seeds**, **M1,M2** (i.e., query-local changes), **M3,M4** (i.e., physical changes), and **All**. We limit the study to the most performant tuner Proto-X and evaluate three scenarios from the same historical DSB workload: **Exact From History**, **50% Parameter Drift**, and **50% Template Drift**. We run four trials for each variation *without* further refinement by Proto-X. We plot the mean performance along with a 95% confidence interval.



(a) Exact From History

Fig. 19. Input Data – The DBMS's performance achieved by different tuners when transferring the same workload as history. We evaluate whether Booster has access to one artifact or all four artifacts from each tuner. We plot the mean performance with a 95% confidence interval obtained by four trials of each technique *without* further refinement.

As shown in Fig. 18, we find that **M2** and **M3** alone have limited effectiveness due to their limited exploration. **M1** and **M4** are more effective by providing a greater degree of diverse composition opportunities. We also observe that although **M3,M4** (i.e., physical changes) finds better configurations in **Exact**, **M1,M2** (i.e., query-local changes) outperforms in **50% Template Drift**. Booster with **All** outputs the best configuration by leveraging all opportunities.

8.6 Input Data

Recall from Sec. 7.1 that Booster ingests artifacts from all four tuner trials. We next measure the efficacy of Booster when only using one artifact or all four artifacts for each tuner. We run four trials for each variation *without* further refinement. We plot the mean performance along with a 95% confidence interval.

As shown in Fig. 19, Booster with **1-Input** performs worse than the **4-Input** for Proto-X and UniTune. By contrast, Booster with **1-Input** performs better for λ-Tune+AS. In the case of Proto-X and UniTune, providing Booster with more data allows it to leverage query-level insights across a broader scope and avoid being overly restricted by a single trial's local optima. For λ-Tune+AS, we find that Booster with **1-Input** augments its prompt with reference QConfigs from more diverse query templates (Sec. 4.2) that lead to more performant outcomes. We defer selecting the number of QConfigs to enrich the prompt based on each query to future work.

9 Related Work

We now discuss existing autonomous DBMS research, focusing on forecasting, behavior modeling, tuning frameworks, representations, natural language debugging, and learned components.

Forecasting: Existing literature focuses on predicting future queries and loads for DBMS optimization. These techniques include arrival rates [56], parameterized queries [34], and identifying ongoing workload drifts [48, 107]. Once these techniques identify a drift, a human operator gathers the new workload, target DBMS deployment, and tuner and provides them to Booster to accelerate the tuner's adaptation to the new environment.

Behavior Modeling: Modeling aims to produce accurate models for inferring the performance of the DBMS [53, 57, 74]. Behavior models can substitute for actual query execution and for speculating on the impact of different configurations. Furthermore, Booster generates substantial DBMS telemetry during operation, which an operator can use to create and refine these models.

Tuning Frameworks: The literature is rich in tuners that target a single DBMS aspect: resource-tuning [1, 38, 82], capacity planning [10, 16], physical design [8, 18, 37, 75, 89, 105], and query tuning [11, 58]. Recent research has focused on tuning over multiple aspects [29, 86, 96, 98].

Representations: This work focuses on deriving a query [102] or workload [85] representation that is conducive to downstream tasks (e.g., behavior models, tuning). These create models that output discriminative vector representations (e.g., embeddings) based on inputs, such as query plans, workload properties, and DBMS statistics. These could optionally be used by Booster to select relevant historical experiences.

Natural Language Debugging: With advances in LLMs, recent work has also focused on debugging SQL issues through a natural language interface. These range from natural language to SQL techniques [26, 101], root cause analysis [66], and debugging user problems [87, 106]. These techniques primarily focus on designing retrieval mechanisms (e.g., ranking functions, fine-tuned embedders) for selecting the top-k chunks to provide to the LLM based on the question.

Learned Components: These are traditional DBMS subsystems augmented with machine learning. Existing work has focused on layouts [25], data structures [31], query processing [73], and query optimization [9, 50, 58]. Other research in this area has focused on learned cardinality estimation [62, 84], due to its impact on query optimizer plan quality [44].

10 Conclusion

Despite advances in tuners' ability to find performant configurations, existing tuners remain unable to adapt to environment changes (e.g., workload drifts, cross-schema transfers) due to their design. To remedy this, we present the Booster framework that exploits query-level insights from history to assist tuners in adapting. Booster organizes historical artifacts into structured insights, obtains query-level configurations by prompting an LLM with relevant experiences, and composes them into a holistic configuration with beam search. We evaluate Booster's ability to assist state-of-the-art cost-/ML-/LLM-based tuners in adapting to new environments for OLAP workloads on PostgreSQL. Compared to the alternative of continuing to tune from historical configurations, Booster assists tuners in finding configurations that improve DBMS performance up to 74% in up to 4.7× less time.

Acknowledgments

This work was supported (in part) by Google DAPA Research Grants, the CMU-DB Industry Affiliates Program, and the MongoDB PhD Fellowship.

References

[1] 2022. PGTune – PostgreSQL configuration wizard. https://github.com/gregs1104/pgtune.

[2] 2022. TPC-H. https://www.tpc.org/tpch.

[3] 2023. HypoPG. https://hypopg.readthedocs.io/.

[4] 2023. pg_hint_plan. https://github.com/17zhangw/pg_hint_plan/tree/parallel_patch.

[5] 2025. OpenRouter Llama 3.1 8B Instruct. https://openrouter.ai/meta-llama/llama-3.1-8b-instruct.

[6] 2025. voyage-3-large. https://blog.voyageai.com/2025/01/07/voyage-3-large/.

[7] 2025. Voyage AI Pricing. https://docs.voyageai.com/docs/pricing.

[8] Rafi Ahmed, Randall Bello, Andrew Witkowski, and Praveen Kumar. 2020. Automated Generation of Materialized Views in Oracle. *Proc. VLDB Endow.* 13, 12 (aug 2020), 3046–3058. https://doi.org/10.14778/3415478.3415533

[9] Peter Akioyamen, Zixuan Yi, and Ryan Marcus. 2024. The Unreasonable Effectiveness of LLMs for Query Optimization. arXiv:2411.02862 [cs.DB] https://arxiv.org/abs/2411.02862

[10] Remmelt Ammerlaan, Gilbert Antonius, Marc Friedman, H M Sajjad Hossain, Alekh Jindal, Peter Orenberg, Hiren Patel, Shi Qiao, Vijay Ramani, Lucas Rosenblatt, Abhishek Roy, Irene Shaffer, Soundarajan Srinivasan, and Markus Weimer. 2021. PerfGuard: Deploying ML-for-Systems without Performance Regressions, Almost! *Proc. VLDB Endow.* 14, 13 (sep 2021), 3362–3375. https://doi.org/10.14778/3484224.3484233

[11] Christoph Anneser, Nesime Tatbul, David Cohen, Zhenggang Xu, Prithviraj Pandian, Nikolay Laptev, and Ryan Marcus. 2023. AutoSteer: Learned Query Optimization for Any SQL Database. *Proc. VLDB Endow.* 16, 12 (sep 2023), 3515–3527. https://doi.org/10.14778/3611540.3611544

[12] Ilias Azizi, Karima Echihabi, and Themis Palpanas. 2023. ELPIS: Graph-Based Similarity Search for Scalable Data Science. *Proc. VLDB Endow.* 16, 6 (Feb. 2023), 1548–1559. https://doi.org/10.14778/3583140.3583166

[13] Rico Bergmann, Claudio Hartmann, Dirk Habich, and Wolfgang Lehner. 2025. An Elephant Under the Microscope: Analyzing the Interaction of Optimizer Components in PostgreSQL. *Proc. ACM Manag. Data* 3, 1, Article 9 (Feb. 2025), 28 pages. https://doi.org/10.1145/3709659

[14] Alexander Bianchi, Andrew Chai, Vincent Corvinelli, Parke Godfrey, Jarek Szlichta, and Calisto Zuzarte. 2024. Db2une: Tuning Under Pressure via Deep Learning. *Proc. VLDB Endow.* 17, 12 (Aug. 2024), 3855–3868. https://doi.org/10.14778/3685800.3685811

[15] Nicolas Bruno and Surajit Chaudhuri. 2005. Automatic physical database tuning: a relaxation-based approach. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data* (Baltimore, Maryland) *(SIGMOD '05)*. Association for Computing Machinery, New York, NY, USA, 227–238. https://doi.org/10.1145/1066157.1066184

[16] Joyce Cahoon, Wenjing Wang, Yiwen Zhu, Katherine Lin, Sean Liu, Raymond Truong, Neetu Singh, Chengcheng Wan, Alexandra Ciortea, Sreraman Narasimhan, and Subru Krishnan. 2022. Doppler: Automated SKU Recommendation in Migrating SQL Workloads to the Cloud. *Proc. VLDB Endow.* 15, 12 (aug 2022), 3509–3521. https://doi.org/10.14778/3554821.3554840

[17] Surajit Chaudhuri and Vivek Narasayya. 1998. AutoAdmin "what-If" Index Analysis Utility. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data* (Seattle, Washington, USA) *(SIGMOD '98)*. Association for Computing Machinery, New York, NY, USA, 367–378. https://doi.org/10.1145/276304.276337

[18] Surajit Chaudhuri and Vivek Narasayya. 2020. Anytime Algorithm of Database Tuning Advisor for Microsoft SQL Server. (June 2020). https://www.microsoft.com/en-us/research/publication/anytime-algorithm-of-database-tuning-advisor-for-microsoft-sql-server/

[19] Sibei Chen, Ju Fan, Bin Wu, Nan Tang, Chao Deng, Pengyi Wang, Ye Li, Jian Tan, Feifei Li, Jingren Zhou, and Xiaoyong Du. 2025. Automatic Database Configuration Debugging using Retrieval-Augmented Language Models. *Proc. ACM Manag. Data* 3, 1, Article 13 (Feb. 2025), 27 pages. https://doi.org/10.1145/3709663

[20] Tianyi Chen, Jun Gao, Hedui Chen, and Yaofeng Tu. 2023. LOGER: A Learned Optimizer Towards Generating Efficient and Robust Query Execution Plans. *Proc. VLDB Endow.* 16, 7 (March 2023), 1777–1789. https://doi.org/10.14778/3587136.3587150

[21] Xu Chen, Haitian Chen, Zibo Liang, Shuncheng Liu, Jinghong Wang, Kai Zeng, Han Su, and Kai Zheng. 2023. LEON: A New Framework for ML-Aided Query Optimization. *Proc. VLDB Endow.* 16, 9 (May 2023), 2261–2273. https://doi.org/10.14778/3598581.3598597

[22] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. arXiv:2205.14135 [cs.LG] https://arxiv.org/abs/2205.14135

[23] Bailu Ding, Surajit Chaudhuri, Johannes Gehrke, and Vivek Narasayya. 2021. DSB: a decision support benchmark for workload-driven and traditional database systems. *Proc. VLDB Endow.* 14, 13 (sep 2021), 3376–3388. https://doi.org/10.14778/3484224.3484234

[24] Bailu Ding, Sudipto Das, Ryan Marcus, Wentao Wu, Surajit Chaudhuri, and Vivek R. Narasayya. 2019. AI Meets AI: Leveraging Query Executions to Improve Index Recommendations. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York,

NY, USA, 1241–1258. https://doi.org/10.1145/3299869.3324957

[25] 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 *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 418–431. https://doi.org/10.1145/3448016.3457270

[26] Ju Fan, Zihui Gu, Songyue Zhang, Yuxin Zhang, Zui Chen, Lei Cao, Guoliang Li, Samuel Madden, Xiaoyong Du, and Nan Tang. 2024. Combining Small Language Models and Large Language Models for Zero-Shot NL2SQL. *Proc. VLDB Endow.* 17, 11 (July 2024), 2750–2763. https://doi.org/10.14778/3681954.3681960

[27] Kai Franz, Samuel I Arch, Denis Hirn, Torsten Grust, Todd Mowry, and Andrew Pavlo. 2024. Dear User-Defined Functions, Inlining isn't working out so great for us. Let's try batching to make our relationship work. Sincerely, SQL. In *CIDR 2024, Conference on Innovative Data Systems Research*.

[28] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Meng Wang, and Haofen Wang. 2024. Retrieval-Augmented Generation for Large Language Models: A Survey. arXiv:2312.10997 [cs.CL] https://arxiv.org/abs/2312.10997

[29] Victor Giannakouris and Immanuel Trummer. 2025. λ-Tune: Harnessing Large Language Models for Automated Database System Tuning. *Proc. ACM Manag. Data* 3, 1, Article 2 (Feb. 2025), 26 pages. https://doi.org/10.1145/3709652

[30] Aaron Grattafiori et al. 2024. The Llama 3 Herd of Models. arXiv:2407.21783 [cs.AI] https://arxiv.org/abs/2407.21783

[31] Tu Gu, Kaiyu Feng, Gao Cong, Cheng Long, Zheng Wang, and Sheng Wang. 2023. The RLR-Tree: A Reinforcement Learning Based R-Tree for Spatial Data. *Proc. ACM Manag. Data* 1, 1, Article 63 (may 2023), 26 pages. https://doi.org/10.1145/3588917

[32] Roman Heinrich, Manisha Luthra, Johannes Wehrstein, Harald Kornmayer, and Carsten Binnig. 2025. How Good are Learned Cost Models, Really? Insights from Query Optimization Tasks. *Proc. ACM Manag. Data* 3, 3, Article 172 (June 2025), 27 pages. https://doi.org/10.1145/3725309

[33] Benjamin Hilprecht and Carsten Binnig. 2022. Zero-Shot Cost Models for out-of-the-Box Learned Cost Prediction. *Proc. VLDB Endow.* 15, 11 (jul 2022), 2361–2374. https://doi.org/10.14778/3551793.3551799

[34] Hanxian Huang, Tarique Siddiqui, Rana Alotaibi, Carlo Curino, Jyoti Leeka, Alekh Jindal, Jishen Zhao, Jesús Camacho-Rodríguez, and Yuanyuan Tian. 2024. Sibyl: Forecasting Time-Evolving Query Workloads. *Proceedings of the ACM on Management of Data* 2, 1 (March 2024), 1–27. https://doi.org/10.1145/3639308

[35] Xinmei Huang, Haoyang Li, Jing Zhang, Xinxin Zhao, Zhiming Yao, Yiyan Li, Tieying Zhang, Jianjun Chen, Hong Chen, and Cuiping Li. 2025. E2ETune: End-to-End Knob Tuning via Fine-tuned Generative Language Model. arXiv:2404.11581 [cs.AI] https://arxiv.org/abs/2404.11581

[36] Bowen Jin, Jinsung Yoon, Jiawei Han, and Sercan O. Arik. 2024. Long-Context LLMs Meet RAG: Overcoming Challenges for Long Inputs in RAG. arXiv:2410.05983 [cs.CL] https://arxiv.org/abs/2410.05983

[37] Andrew Kane. [n.d.]. Introducing Dexter, the Automatic Indexer for Postgres — ankane. https://medium.com/@ankane/introducing-dexter-the-automatic-indexer-for-postgres-5f8fa8b28f27.

[38] Konstantinos Kanellis, Cong Ding, Brian Kroth, Andreas C. Müller, Carlo Curino, and Shivaram Venkataraman. 2022. LlamaTune: Sample-Efficient DBMS Configuration Tuning. In *VLDB 2022*. https://www.microsoft.com/en-us/research/publication/llamatune-sample-efficient-dbms-configuration-tuning/

[39] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. In *Proceedings of the 36th International Conference on Neural Information Processing Systems* (New Orleans, LA, USA) *(NIPS '22)*. Curran Associates Inc., Red Hook, NY, USA, Article 1613, 15 pages.

[40] Jan Kossmann, Stefan Halfpap, Marcel Jankrift, and Rainer Schlosser. 2020. Magic Mirror in My Hand, Which is the Best in the Land? An Experimental Evaluation of Index Selection Algorithms. *Proc. VLDB Endow.* 13, 12 (jul 2020), 2382–2395. https://doi.org/10.14778/3407790.3407832

[41] Jan Kossmann, Alexander Kastius, and Rainer Schlosser. 2022. SWIRL: Selection of Workload-aware Indexes using Reinforcement Learning. In *Proceedings of the 25th International Conference on Extending Database Technology, EDBT 2022, Edinburgh, UK, March 29 - April 1, 2022*, Julia Stoyanovich, Jens Teubner, Paolo Guagliardo, Milos Nikolic, Andreas Pieris, Jan Mühlig, Fatma Özcan, Sebastian Schelter, H. V. Jagadish, and Meihui Zhang (Eds.). OpenProceedings.org, 2:155–2:168. https://doi.org/10.48786/edbt.2022.06

[42] Jan Kossmann and Rainer Schlosser. 2020. Self-driving database systems: a conceptual approach. *Distrib. Parallel Databases* 38, 4 (Dec. 2020), 795–817. https://doi.org/10.1007/s10619-020-07288-w

[43] Leonardo Kuffo, Elena Krippner, and Peter Boncz. 2025. PDX: A Data Layout for Vector Similarity Search. arXiv:2503.04422 [cs.DB] https://arxiv.org/abs/2503.04422

[44] Kukjin Lee, Anshuman Dutt, Vivek Narasayya, and Surajit Chaudhuri. 2023. Analyzing the Impact of Cardinality Estimation on Execution Plans in Microsoft SQL Server. *Proc. VLDB Endow.* 16, 11 (aug 2023), 2871–2883. https://doi.org/10.14778/3611479.3611494

[45] Claude Lehmann, Pavel Sulimov, and Kurt Stockinger. 2024. Is Your Learned Query Optimizer Behaving As You Expect? A Machine Learning Perspective. *Proc. VLDB Endow.* 17, 7 (March 2024), 1565–1577. https://doi.org/10.14778/3654621.3654625

[46] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9, 3 (nov 2015), 204–215. https://doi.org/10.14778/2850583.2850594

[47] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 9459–9474. https://proceedings.neurips.cc/paper_files/paper/2020/file/6b493230205f780e1bc26945df7481e5-Paper.pdf

[48] Beibin Li, Yao Lu, and Srikanth Kandula. 2022. Warper: Efficiently Adapting Learned Cardinality Estimators to Data and Workload Drifts. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 1920–1933. https://doi.org/10.1145/3514221.3526179

[49] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. 2019. QTune: A Query-Aware Database Tuning System with Deep Reinforcement Learning. *Proc. VLDB Endow.* 12, 12 (aug 2019), 2118–2130. https://doi.org/10.14778/3352063.3352129

[50] Yifan Li, Xiaohui Yu, Nick Koudas, Shu Lin, Calvin Sun, and Chong Chen. 2023. DbET: Execution Time Distribution-Based Plan Selection. *Proc. ACM Manag. Data* 1, 1, Article 31 (may 2023), 26 pages. https://doi.org/10.1145/3588711

[51] Zhaodonghui Li, Haitao Yuan, Huiming Wang, Gao Cong, and Lidong Bing. 2024. LLM-R2: A Large Language Model Enhanced Rule-Based Rewrite System for Boosting Query Efficiency. *Proc. VLDB Endow.* 18, 1 (Sept. 2024), 53–65. https://doi.org/10.14778/3696435.3696440

[52] Wan Shen Lim, Matthew Butrovich, William Zhang, Andrew Crotty, Lin Ma, Peijing Xu, Johannes Gehrke, and Andrew Pavlo. 2023. Database Gyms. In *CIDR 2023, Conference on Innovative Data Systems Research*.

[53] Wan Shen Lim, Lin Ma, William Zhang, Matthew Butrovich, Samuel Arch, and Andrew Pavlo. 2024. Hit the Gym: Accelerating Query Execution to Efficiently Bootstrap Behavior Models for Self-Driving Database Management Systems. *Proc. VLDB Endow.* 17, 11 (July 2024), 3680–3693. https://doi.org/10.14778/3681954.3682030

[54] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the Middle: How Language Models Use Long Contexts. *Transactions of the Association for Computational Linguistics* 12 (2024), 157–173. https://doi.org/10.1162/tacl_a_00638

[55] Lin Ma, Bailu Ding, Sudipto Das, and Adith Swaminathan. 2020. Active Learning for ML Enhanced Database Systems. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 175–191. https://doi.org/10.1145/3318464.3389768

[56] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J. Gordon. 2018. Query-based Workload Forecasting for Self-Driving Database Management Systems. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 631–645. https://doi.org/10.1145/3183713.3196908

[57] Lin Ma, William Zhang, Jie Jiao, Wuwen Wang, Matthew Butrovich, Wan Shen Lim, Prashanth Menon, and Andrew Pavlo. 2021. MB2: Decomposed Behavior Modeling for Self-Driving Database Management Systems. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 1248–1261. https://doi.org/10.1145/3448016.3457276

[58] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2021. Bao: Making Learned Query Optimization Practical. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 1275–1288. https://doi.org/10.1145/3448016.3452838

[59] Ryan Marcus and Olga Papaemmanouil. 2019. Plan-Structured Deep Neural Network Models for Query Performance Prediction. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1733–1746.

[60] Shervin Minaee, Tomas Mikolov, Narjes Nikzad, Meysam Chenaghlu, Richard Socher, Xavier Amatriain, and Jianfeng Gao. 2025. Large Language Models: A Survey. arXiv:2402.06196 [cs.CL] https://arxiv.org/abs/2402.06196

[61] Niklas Muennighoff, Nouamane Tazi, Loïc Magne, and Nils Reimers. 2023. MTEB: Massive Text Embedding Benchmark. arXiv:2210.07316 [cs.CL] https://arxiv.org/abs/2210.07316

[62] Parimarjan Negi, Ziniu Wu, Andreas Kipf, Nesime Tatbul, Ryan Marcus, Sam Madden, Tim Kraska, and Mohammad Alizadeh. 2023. Robust Query Driven Cardinality Estimation under Changing Workloads. *Proc. VLDB Endow.* 16, 6 (apr 2023), 1520–1533. https://doi.org/10.14778/3583140.3583164

[63] OpenAI. 2024. GPT-4o System Card. arXiv:2410.21276 [cs.CL] https://arxiv.org/abs/2410.21276

[64] OpenAI. 2024. New embedding models and API updates. https://openai.com/index/new-embedding-models-and-api-updates/.

[65] OpenAI. 2025. OpenAI o3-mini. https://openai.com/index/openai-o3-mini/.

[66] Biao Ouyang, Yingying Zhang, Hanyin Cheng, Yang Shu, Chenjuan Guo, Bin Yang, Qingsong Wen, Lunting Fan, and Christian S. Jensen. 2025. RCRank: Multimodal Ranking of Root Causes of Slow Queries in Cloud Database Systems. *Proc. VLDB Endow.* 18, 4 (May 2025), 1169–1182. https://doi.org/10.14778/3717755.3717774

[67] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd Mowry, Matthew Perron, Ian Quah, Siddharth Santurkar, Anthony Tomasic, Skye Toor, Dana Van Aken, Ziqi Wang, Yingjun Wu, Ran Xian, and Tieying Zhang. 2017. Self-Driving Database Management Systems. In *CIDR 2017, Conference on Innovative Data Systems Research*.

[68] Andrew Pavlo, Matthew Butrovich, Lin Ma, Prashanth Menon, Wan Shen Lim, Dana Van Aken, and William Zhang. 2021. Make Your Database System Dream of Electric Sheep: Towards Self-Driving Operation. *Proc. VLDB Endow.* 14, 12 (July 2021), 3211–3221. https://doi.org/10.14778/3476311.3476411

[69] Judea Pearl. 1984. *Heuristics: intelligent search strategies for computer problem solving*. Addison-Wesley Longman Publishing Co., Inc., USA.

[70] Deepak Babu Piskala, Vijay Raajaa, Sachin Mishra, and Bruno Bozza. 2024. OPTIROUTE Dynamic LLM Routing and Selection Based on User Preferences: Balancing Performance, Cost, and Ethics. *International Journal of Computer Applications* 186, 51 (Nov. 2024), 1–7. https://doi.org/10.5120/ijca2024924172

[71] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. DeepSpeed: System Optimizations Enable Training Deep Learning Models with Over 100 Billion Parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (Virtual Event, CA, USA) *(KDD '20)*. Association for Computing Machinery, New York, NY, USA, 3505–3506. https://doi.org/10.1145/3394486.3406703

[72] Aniket Rege, Aditya Kusupati, Sharan Ranjit S, Alan Fan, Qingqing Cao, Sham Kakade, Prateek Jain, and Ali Farhadi. 2023. AdANNS: A Framework for Adaptive Semantic Search. In *Advances in Neural Information Processing Systems*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (Eds.), Vol. 36. Curran Associates, Inc., 76311–76335. https://proceedings.neurips.cc/paper_files/paper/2023/file/f062da1973ac9ac61fc6d44dd7fa309f-Paper-Conference.pdf

[73] Ibrahim Sabek and Tim Kraska. 2023. The Case for Learned In-Memory Joins. *Proc. VLDB Endow.* 16, 7 (may 2023), 1749–1762. https://doi.org/10.14778/3587136.3587148

[74] Jiachen Shi, Gao Cong, and Xiao-Li Li. 2022. Learned Index Benefits: Machine Learning Based Index Performance Estimation. *Proc. VLDB Endow.* 15, 13 (sep 2022), 3950–3962. https://doi.org/10.14778/3565838.3565848

[75] Tarique Siddiqui, Wentao Wu, Vivek Narasayya, and Surajit Chaudhuri. 2022. DISTILL: Low-Overhead Data-Driven Techniques for Filtering and Costing Indexes for Scalable Index Tuning. *Proc. VLDB Endow.* 15, 10 (jun 2022), 2019–2031. https://doi.org/10.14778/3547305.3547309

[76] Adam J. Storm, Christian Garcia-Arellano, Sam S. Lightstone, Yixin Diao, and M. Surendra. 2006. Adaptive Self-tuning Memory in DB2. In *VLDB*. 1081–1092.

[77] Pavle Subotić, Herbert Jordan, Lijun Chang, Alan Fekete, and Bernhard Scholz. 2018. Automatic index selection for large-scale datalog computation. *Proc. VLDB Endow.* 12, 2 (Oct. 2018), 141–153. https://doi.org/10.14778/3282495.3282500

[78] Tarun Suresh, Revanth Gangi Reddy, Yifei Xu, Zach Nussbaum, Andriy Mulyar, Brandon Duderstadt, and Heng Ji. 2025. CoRNStack: High-Quality Contrastive Data for Better Code Retrieval and Reranking. In *The Thirteenth International Conference on Learning Representations*. https://openreview.net/forum?id=iyJOUELYir

[79] Xiu Tang, Wenhao Liu, Sai Wu, Chang Yao, Gongsheng Yuan, Shanshan Ying, and Gang Chen. 2024. QueryArtisan: Generating Data Manipulation Codes for Ad-hoc Analysis in Data Lakes. *Proc. VLDB Endow.* 18, 2 (2024), 108–116. https://www.vldb.org/pvldb/vol18/p108-yao.pdf

[80] Jeffrey Tao, Natalie Maus, Haydn Jones, Yimeng Zeng, Jacob R. Gardner, and Ryan Marcus. 2025. Learned Offline Query Planning via Bayesian Optimization. arXiv:2502.05256 [cs.DB] https://arxiv.org/abs/2502.05256

[81] The Transaction Processing Council. 2021. TPC-DS Benchmark (Revision 3.2.0). https://www.tpc.org/TPC_Documents_Current_Versions/pdf/TPC-DS_v3.2.0.pdf.

[82] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. 2017. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 1009–1024. https://doi.org/10.1145/3035918.3064029

[83] Alexander van Renen, Dominik Horn, Pascal Pfeil, Kapil Vaidya, Wenjian Dong, Murali Narayanaswamy, Zhengchun Liu, Gaurav Saxena, Andreas Kipf, and Tim Kraska. 2024. Why TPC is Not Enough: An Analysis of the Amazon Redshift Fleet. *Proc. VLDB Endow.* 17, 11 (July 2024), 3694–3706. https://doi.org/10.14778/3681954.3682031

[84] Fang Wang, Xiao Yan, Man Lung Yiu, Shuai LI, Zunyao Mao, and Bo Tang. 2023. Speeding Up End-to-End Query Execution via Learning-Based Progressive Cardinality Estimation. *Proc. ACM Manag. Data* 1, 1, Article 28 (may 2023), 25 pages. https://doi.org/10.1145/3588708

[85] Jiaqi Wang, Tianyi Li, Anni Wang, Xiaoze Liu, Lu Chen, Jie Chen, Jianye Liu, Junyang Wu, Feifei Li, and Yunjun Gao. 2023. Real-Time Workload Pattern Analysis for Large-Scale Cloud Databases. *Proc. VLDB Endow.* 16, 12 (sep 2023), 3689–3701. https://doi.org/10.14778/3611540.3611557

[86] Junxiong Wang, Immanuel Trummer, and Debabrota Basu. 2021. UDO: Universal Database Optimization Using Reinforcement Learning. *Proc. VLDB Endow.* 14, 13 (sep 2021), 3402–3414. https://doi.org/10.14778/3484224.3484236

[87] Pengyi Wang, Sibei Chen, Ju Fan, Bin Wu, Nan Tang, and Jian Tan. 2025. Andromeda: Debugging Database Performance Issues with Retrieval-Augmented Large Language Models. In *Companion of the 2025 International Conference on Management of Data* (Berlin, Germany) *(SIGMOD/PODS '25)*. Association for Computing Machinery, New York, NY, USA, 243–246. https://doi.org/10.1145/3722212.3725080

[88] Xiaoying Wang, Changbo Qu, Weiyuan Wu, Jiannan Wang, and Qingqing Zhou. 2021. Are We Ready for Learned Cardinality Estimation? *Proc. VLDB Endow.* 14, 9 (may 2021), 1640–1654. https://doi.org/10.14778/3461535.3461552

[89] Zijia Wang, Haoran Liu, Chen Lin, Zhifeng Bao, Guoliang Li, and Tianqing Wang. 2024. Leveraging Dynamic and Heterogeneous Workload Knowledge to Boost the Performance of Index Advisors. *Proc. VLDB Endow.* 17, 7 (March 2024), 1642–1654. https://doi.org/10.14778/3654621.3654631

[90] Wentao Wu, Chi Wang, Tarique Siddiqui, Junxiong Wang, Vivek Narasayya, Surajit Chaudhuri, and Philip A. Bernstein. 2022. Budget-Aware Index Tuning with Reinforcement Learning. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 1528–1541. https://doi.org/10.1145/3514221.3526128

[91] Mengyi Yan, Yaoshu Wang, Yue Wang, Xiaoye Miao, and Jianxin Li. 2024. GIDCL: A Graph-Enhanced Interpretable Data Cleaning Framework with Large Language Models. *Proc. ACM Manag. Data* 2, 6, Article 236 (Dec. 2024), 29 pages. https://doi.org/10.1145/3698811

[92] An Yang et al. 2025. Qwen3 Technical Report. arXiv:2505.09388 [cs.CL] https://arxiv.org/abs/2505.09388

[93] Zhiming Yao, Haoyang Li, Jing Zhang, Cuiping Li, and Hong Chen. 2025. A Query Optimization Method Utilizing Large Language Models. arXiv:2503.06902 [cs.DB] https://arxiv.org/abs/2503.06902

[94] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, Minwei Ran, and Zekang Li. 2019. An End-to-End Automatic Cloud Database Tuning System Using Deep Reinforcement Learning. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 415–432. https://doi.org/10.1145/3299869.3300085

[95] Shengyu Zhang, Linfeng Dong, Xiaoya Li, Sen Zhang, Xiaofei Sun, Shuhe Wang, Jiwei Li, Runyi Hu, Tianwei Zhang, Fei Wu, and Guoyin Wang. 2024. Instruction Tuning for Large Language Models: A Survey. arXiv:2308.10792 [cs.CL] https://arxiv.org/abs/2308.10792

[96] William Zhang, Wan Shen Lim, Matthew Butrovich, and Andrew Pavlo. 2024. 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.* 17, 11 (2024), 3373–3387. https://www.vldb.org/pvldb/vol17/p3373-zhang.pdf

[97] Xinyi Zhang, Zhuo Chang, Yang Li, Hong Wu, Jian Tan, Feifei Li, and Bin Cui. 2022. Facilitating Database Tuning with Hyper-Parameter Optimization: A Comprehensive Experimental Evaluation. *Proc. VLDB Endow.* 15, 9 (may 2022), 1808–1821. https://doi.org/10.14778/3538598.3538604

[98] Xinyi Zhang, Zhuo Chang, Hong Wu, Yang Li, Jia Chen, Jian Tan, Feifei Li, and Bin Cui. 2023. A Unified and Efficient Coordinating Framework for Autonomous DBMS Tuning. *Proc. ACM Manag. Data* 1, 2, Article 186 (jun 2023), 26 pages. https://doi.org/10.1145/3589331

[99] Xinyi Zhang, Hong Wu, Yang Li, Zhengju Tang, Jian Tan, Feifei Li, and Bin Cui. 2023. An Efficient Transfer Learning Based Configuration Adviser for Database Tuning. *Proc. VLDB Endow.* 17, 3 (nov 2023), 539–552. https://doi.org/10.14778/3632093.3632114

[100] Yunjia Zhang, Jordan Henkel, Avrilia Floratou, Joyce Cahoon, Shaleen Deep, and Jignesh M. Patel. 2024. ReAcTable: Enhancing ReAct for Table Question Answering. *Proc. VLDB Endow.* 17, 8 (April 2024), 1981–1994. https://doi.org/10.14778/3659437.3659452

[101] Fuheng Zhao, Shaleen Deep, Fotis Psallidas, Avrilia Floratou, Divyakant Agrawal, and Amr El Abbadi. 2025. Sphinteract: Resolving Ambiguities in NL2SQL through User Interaction. *Proc. VLDB Endow.* 18, 4 (May 2025), 1145–1158. https://doi.org/10.14778/3717755.3717772

[102] Yue Zhao, Gao Cong, Jiachen Shi, and Chunyan Miao. 2022. QueryFormer: A Tree Transformer Model for Query Plan Representation. *Proc. VLDB Endow.* 15, 8 (apr 2022), 1658–1670. https://doi.org/10.14778/3529337.3529349

[103] Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, and Zheyan Luo. 2024. LlamaFactory: Unified Efficient Fine-Tuning of 100+ Language Models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, Yixin Cao, Yang Feng, and Deyi Xiong (Eds.). Association for Computational Linguistics, Bangkok, Thailand, 400–410. https://doi.org/10.18653/v1/2024.acl-demos.38

[104] Xuanhe Zhou, Lianyuan Jin, Ji Sun, Xinyang Zhao, Xiang Yu, Jianhua Feng, Shifu Li, Tianqing Wang, Kun Li, and Luyang Liu. 2021. DBMind: a self-driving platform in openGauss. *Proc. VLDB Endow.* 14, 12 (July 2021), 2743–2746. https://doi.org/10.14778/3476311.3476334

[105] Xuanhe Zhou, Guoliang Li, Jianhua Feng, Luyang Liu, and Wei Guo. 2023. Grep: A Graph Learning Based Database Partitioning System. *Proc. ACM Manag. Data* 1, 1, Article 94 (may 2023), 24 pages. https://doi.org/10.1145/3588948

[106] Xuanhe Zhou, Guoliang Li, Zhaoyan Sun, Zhiyuan Liu, Weize Chen, Jianming Wu, Jiesi Liu, Ruohang Feng, and Guoyang Zeng. 2024. D-Bot: Database Diagnosis System using Large Language Models. *Proc. VLDB Endow.* 17, 10 (June 2024), 2514–2527. https://doi.org/10.14778/3675034.3675043

[107] Jiaqi Zhu, Shaofeng Cai, Fang Deng, Beng Chin Ooi, and Wenqiao Zhang. 2023. METER: A Dynamic Concept Adaptation Framework for Online Anomaly Detection. *Proc. VLDB Endow.* 17, 4 (Dec. 2023), 794–807. https://doi.org/10.14778/3636218.3636233

Received July 2025; revised October 2025; accepted October 2025