The Holon Approach for Simultaneously Tuning Multiple Components in a Self-Driving Database Management System with Machine Learning via Synthesized Proto-Actions

William Zhang
wz2@cs.cmu.edu
Carnegie Mellon University

Wan Shen Lim
wanshenl@cs.cmu.edu
Carnegie Mellon University

Matthew Butrovich
mbutrovi@cs.cmu.edu
Carnegie Mellon University

Andrew Pavlo
pavlo@cs.cmu.edu
Carnegie Mellon University

ABSTRACT

Existing machine learning (ML) approaches to automatically optimize database management systems (DBMSs) only target a single configuration space at a time (e.g., knobs, query hints, indexes). Simultaneously tuning multiple configuration spaces is challenging due to the combined space's complexity. Previous tuning methods work around this by sequentially tuning individual spaces with a pool of tuners. However, these approaches struggle to coordinate their tuners and get stuck in local optima.

This paper presents the Proto-X framework that holistically tunes multiple configuration spaces. The key idea of Proto-X is to identify similarities across multiple spaces, encode them in a high-dimensional model, and then synthesize "proto-actions" to navigate the organized space for promising configurations. We evaluate Proto-X against state-of-the-art DBMS tuning frameworks on tuning PostgreSQL for analytical and transactional workloads. By reasoning about configuration spaces that are orders of magnitude more complex than other frameworks (both in terms of quantity and variety), Proto-X discovers configurations that improve PostgreSQL's performance by up to 53% over the next best approach.

PVLDB Reference Format:
William Zhang, Wan Shen Lim, Matthew Butrovich, and Andrew Pavlo. The Holon Approach for Simultaneously Tuning Multiple Components in a Self-Driving Database Management System with Machine Learning via Synthesized Proto-Actions. PVLDB, 17(11): 3373-3387, 2024.
doi:10.14778/3681954.3682007

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/17zhangw/protox.

1 INTRODUCTION

The complexity of DBMSs has increased in tandem with the complexity of applications' workloads. This is evident in the number of configurable options that share subtle interactions: system knobs, query knobs, object knobs (e.g., table/index knobs), and indexes. We refer to each set of such options as a *configuration space*.

Heuristic and cost-based search approaches for knob and index tuning are well-studied [2, 9]. These techniques rely on derived costs via a "what-if" mechanism [20] instead of gathering workload telemetry (e.g., pages fetched). These heuristics struggle when the solution is complex [77] or the costs are erroneous [44].

The last decade saw the rise of using ML to tune a single DBMS configuration space [31, 35, 39, 59, 70, 82, 83]. These approaches pass the DBMS's representation to a model to obtain a suggested configuration, deploy the suggestion, and evaluate the user's objective function (e.g., query latency) to obtain a reward. They then feed the reward back to the model to refine future recommendations.

Tuning individual configuration spaces is inadequate to achieve a fully autonomous (i.e., self-driving [59]) DBMS. A self-driving DBMS leverages its understanding of the system's internals to find an objective-maximizing workload-specific configuration without user intervention. This search requires the self-driving DBMS to select actions (e.g., change a system knob, build an index, alter a query knob/hint) across multiple configuration spaces.

Applying existing techniques directly to the combination of spaces (i.e., the holistic configuration space) suffers from the *curse of dimensionality* [84]. In a holistic space, the landscape of beneficial configurations becomes sparser. Due to this, these approaches degrade and spend too much time evaluating unpromising solutions.

Recent work has proposed using multiple tuners [74, 84] to manage this complexity, with each tuner focusing on a separate configuration space. However, these approaches suffer from the *coordination problem* (i.e., *prisoner's dilemma*) where an individual tuner will never make a locally suboptimal decision even though that might enable a subsequent tuner to find the global optima.

Given this, we introduce the **Proto-X** DBMS tuning framework that handles large solution spaces by exploiting similarities between configurations: *structural* (e.g., similar indexes or knobs) and *objective* (e.g., similar performance). Proto-X uses this similarity to shape the holistic space into neighborhoods and navigates them to find promising configurations. We evaluate Proto-X against other state-of-the-art methods to tune PostgreSQL for OLAP and OLTP workloads. By supporting more diverse spaces (e.g., system/table/index knobs, indexes, and query hints) and orders of magnitude more options, Proto-X discovers configurations up to 54% over the next best approach.

We lay out the remainder of the paper as follows. We provide background into tuning multiple configuration spaces and the ML methods underpinning our technique in Sec. 2. We then provide an overview of Proto-X in Sec. 3, discuss shaping the holistic space in Sec. 4, and describe the agent's action selection process in Sec. 5. We evaluate Proto-X against other techniques in Sec. 6 and present sensitivity studies in Sec. 7.

Proceedings of the VLDB Endowment, Vol. 17, No. 11 ISSN 2150-8097.
doi:10.14778/3681954.3682007

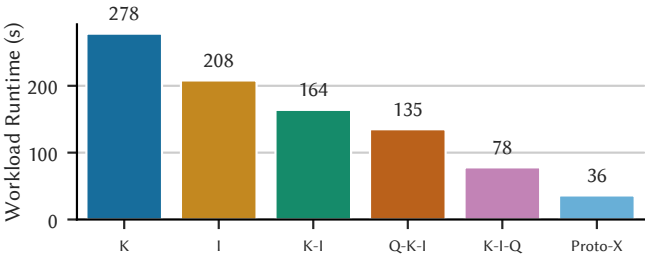


Figure 1: Motivating Example – JOB workload runtime when running a knob tuning agent (K), an index tuning agent (I), and a query knob tuning agent (Q) in isolation, sequentially, and our technique Proto-X that holistically optimizes knobs, indexes, and query knobs/hints for 30h.

2 BACKGROUND

A self-driving DBMS optimizes itself within user-defined constraints (e.g., resource limits) without human intervention [58, 59]. The DBMS first performs workload forecasting to predict future workloads (e.g., time series of SQL queries) [50]. It then builds *behavior models* [51, 55] to estimate the user's objective function (i.e., performance) for a given configuration. The DBMS's *tuning agent* then chooses *actions* to maximize the objective function on the future workload. For example, an action can build an index, change a table knob, or modify a query plan.

Existing work tunes a specific category of DBMS options (i.e., a *configuration space*) in isolation. For instance, different tuners target system knob tuning [70, 82], query knob tuning [13, 53], and physical design [60, 65]. As we now discuss, running these tuners one at a time is not optimal due to interactions between the spaces.

2.1 Sequential Tuning and Coordination

DBMS tuners [74, 84] that support multiple configuration spaces repeatedly invoke the same steps: (1) choose a single space to tune, (2) fix other spaces to their current configuration, and (3) tune the target space for a fixed amount of time. For example, UniTune [84] may tune knobs for an hour, then indexes, and then back to knobs.

As sequential tuning cannot consider actions across spaces and fails to find optimal solutions, deployments incur wasted resources from repeating workloads [78] or undesirable latency [13]. Fig. 1 illustrates the runtime on the JOB workload [44] of the best configuration discovered by sequential methods and our holistic technique Proto-X in 30h. Proto-X finds the best configuration (36s), which executes 42s (53%) faster than that found by the next best method.

We analyze JOB's Q26c to illustrate this problem in more detail. From an initial configuration, we alternate index and query tuners to obtain a search tree of <index set, query knobs> configurations along with Q26c's corresponding performance in Fig. 2. Each tuner locally maximizes its component and prunes suboptimal actions (the red X). This process simplifies the search. However, the pruning prematurely eliminates paths to the global optima <I2,H2>.

2.2 Holistic Optimization and Action Similarity

To our knowledge, no prior approach optimizes the entire configuration space simultaneously due to the resulting holistic space's complexity [47, 65, 84]. For instance, there are at least 2^{46} candidate indexes from TPC-DS [68]. This complexity further compounds when considering other spaces in conjunction, such as query knobs.

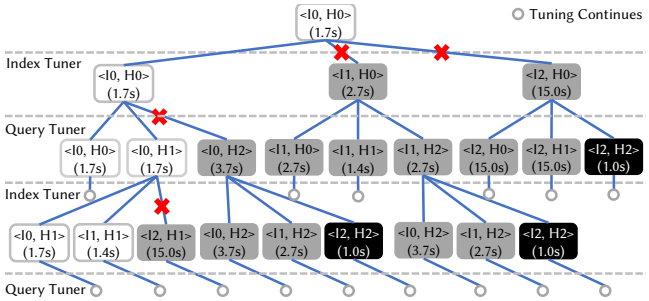


Figure 2: JOB Q26c Sequential Tuning Tree – Search tree explored by sequentially tuning indexes and query knobs. Each node represents a <index set, query knob> configuration with Q26's corresponding performance. The tuners prune branches indicated in red (X). Sequential tuning finds a local optima at <I1,H1> but will not find the global optima in black (<I2,H2>).

Although the number of unique actions in a space is large, many share properties that it may not be necessary to consider each one individually. Two actions are *similar* if they target the same database objects and have roughly the same effect on the agent's objective function. Consider two candidate actions that add indexes to the TPC-H `LINEITEM` table [3]: (1) Cand-A builds an index on `(l_partkey)`, and (2) Cand-B builds an index on `(l_partkey,l_commitdate)`. Both actions have similar expected changes to the DBMS's performance and are structurally similar (i.e., the indexes target the same table and share the first key).

Such similarity allows a tuning agent to infer the performance of one action from a similar action. Rather than running the workload once each for Cand-A and Cand-B, the agent can estimate Cand-B's benefit by only evaluating the workload with Cand-A and vice versa. This similarity suggests that agents should not consider actions alone. Instead, an agent should consider an action and its neighborhood of similar actions together.

2.3 Proto-Actions and Neighborhoods

We combine actions that modify the DBMS's configuration into a *holon* comprised of multiple fields [41, 66]. Each holon field is an independent action corresponding to a configuration space under tuning. We illustrate an example *holon* in Fig. 3 ❶. `K` sets a system knob that sets the buffer pool size (`shared_buffers`); `Q-c` sets a query knob that disables nested loop joins on Q1 of the workload; `Q-e` sets a query hint that forces Q2 to scan `region` in parallel; `I` builds an index on `LINEITEM (l_partkey,l_suppkey)`.

An encoder (e.g., neural network) maps the holon to a point in a multidimensional continuous *latent space*. We shape the latent space by training the encoder to place holons of similar structure or expected performance in nearby points (i.e., close to each other) [52]. The latent space's dimensionality captures a trade-off between the accuracy of holon representations and ease of exploration.

Unlike other tuners that directly suggest deployable actions, Proto-X's agent suggests a latent space point (i.e., a *proto-action*). A proto-action represents a neighborhood of holons with similar structure or performance. Directly decoding the proto-action may not yield a valid holon (e.g., ❷ the proto-action sets a integer knob `shared_buffers` to 519.9). Instead, the agent decodes and ❸ searches the proto-action's neighborhood to obtain a candidate set, from which it selects the most promising holon to evaluate [28].

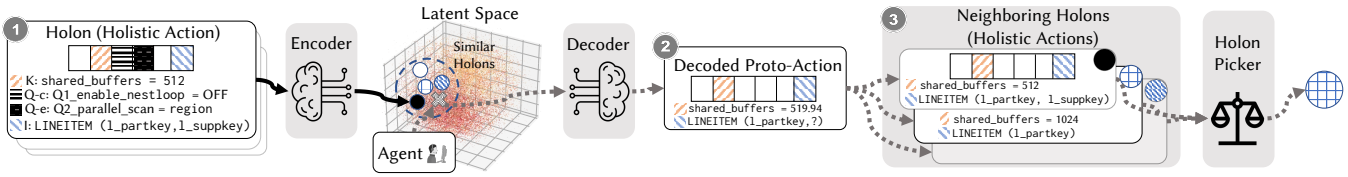


Figure 3: Proto-Actions and Neighborhoods – A holon passes through an encoder (e.g., neural network) to obtain a point (filled-in circle) in latent space where holons of similar performance or structure are nearby. A proto-action is a point (**X**) in latent space chosen by a tuning agent. Decoding the proto-action does not necessarily result in a valid holon. Instead, the agent searches the proto-action's neighborhood for a valid holon and selects the most promising one.

3 OVERVIEW

We present the architecture of the Proto-X automated tuning framework in Fig. 4. Like other offline tuning tools [51, 53, 70, 74, 82, 84], Proto-X assumes access to a representative or historical workload sample [22, 78], an isolated environment for tuning [47], and infrastructure (e.g., proxy) for applying query options, such as those exposed by PostgreSQL's *pg_hint_plan* [5]. Proto-X operates in two phases. First, it creates the latent space based on the DBMS schema, sample workload, and target configuration spaces. Then, in the second phase, Proto-X exploits the created latent space to tune the DBMS. We now describe each phase in more detail.

3.1 Phase I: Latent Space Creation

Proto-X begins by collecting each configuration space's metadata: (1) *system knobs*, (2) *table knobs*, (3) *indexes and per-index knobs*, (4) *query knobs*, and (5) *query hints*. A query knob alters the DBMS's behavior by changing a system knob for a specific query (e.g., hash join memory allocation). A query hint injects commands to control how the DBMS generates a plan (e.g., index scan for a table).

Proto-X obtains the tunable system knobs either from the user, through a DBMS-specific method, or from externally derived knob constraints [85]. For each system knob, Proto-X obtains the min/max values and whether the knob should be restricted to a limited value range [59] (e.g., quantization). Next, it connects to the DBMS to obtain the database schema to identify candidate table knobs. Proto-X defines its candidate index domain by extracting all referenced attributes from the predicates of the user's workload [65]and obtains tunable per-index knobs from the user's specification.

Lastly, Proto-X parses the workload to identify query knobs and hints. For each query, Proto-X identifies (1) query knobs based on a user-supplied list or by mining the system knobs for applicable knobs, (2) a query hint for each referenced table that forces a specific access method, and (3) a query hint specified once per-query that forces a parallel scan on a selected table. For example, TPC-H [3]'s Q14 accesses the `LINEITEM` and `PART` tables, and thus Proto-X generates the `Q14_lineitem_scan` and `Q14_part_scan` hints that control whether the DBMS should use an index scan for those tables. Proto-X also identifies a `Q14_Parallel` hint. Setting this hint forces the DBMS to scan the referenced table in parallel.

After obtaining metadata, in Fig. 4, Proto-X ❶ samples a batch of holons and their estimated benefits for the target workload. ❷ Using the batch, Proto-X trains the latent space to place holons of similar benefit at nearby locations. For example, if an index `t (a,b,c)` has approximately the same benefit as an index `t (a,c,b)`, their corresponding actions will be close to each other in the latent space. Proto-X repeats this process several times to refine the latent space. We elaborate further in Sec. 4.

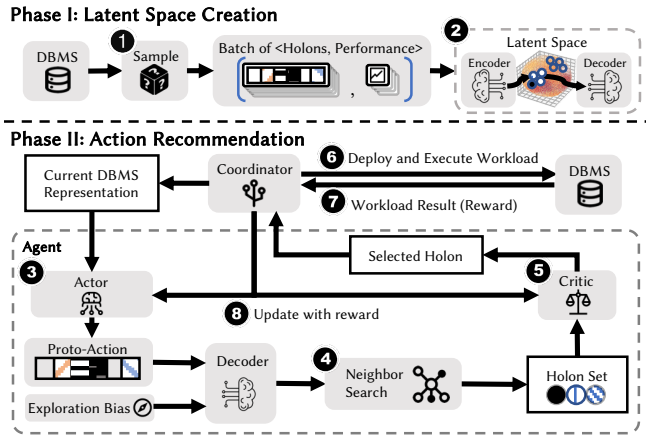


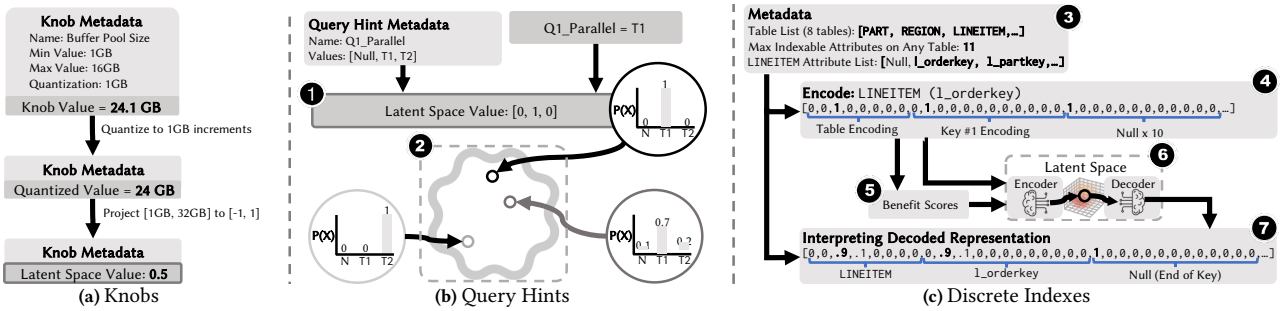
Figure 4: Architecture – An overview of the two phases of Proto-X. In Phase I, Proto-X creates the necessary latent space. In Phase II, Proto-X recommends holons to optimize the DBMS.

3.2 Phase II: Holon Recommendation

Proto-X then instantiates an agent to tune the user's DBMS. The coordinator handles interfacing with the DBMS, from deploying holons to evaluating the user's objective function on the sample workload. The agent is based on the *Wolpertinger* Architecture [28], which uses an actor network to emit a proto-action, refines the proto-action to obtain a neighborhood of candidate holons, and employs a critic network to select the most promising holon.

In Fig. 4 ❸, Phase II starts with the agent passing the current DBMS representation through its actor network to obtain a proto-action. The representation is either the deployed configuration (e.g., knobs, indexes) or internal metrics data (e.g., tuples processed, pages read) [82]. The agent then augments the proto-action with an exploration bias that Proto-X tweaks with small adjustments over time to encourage the exploration and coverage of less promising regions. We provide a further analysis of this in Sec. 7.6.

The agent then decodes the augmented proto-action, searches its neighborhood (see Sec. 5.2) to obtain a holon set, and ❺ uses its critic network to select the most promising candidate. ❻ The coordinator deploys the selected holon, runs the sample workload, and ❼ evaluates the user's objective function to obtain a reward. ❽ The coordinator updates the actor and critic networks with the reward and advances the agent to the next tuning step. The agent and coordinator repeat this process until the agent reaches an illegal state (e.g., an invalid configuration) or a fixed number of steps has elapsed. At this point, the coordinator resets the environment to the initial or a previously discovered configuration before resuming.



(a) Knobs (b) Query Hints (c) Discrete Indexes

Figure 5: Latent Space Creation – Illustrates how Proto-X defines and creates the latent space for all supported configuration spaces depending on the space's type. Knobs includes system knobs, table knobs, and query knobs. In Fig. 5b, "P(X)" refers to the probability function. We illustrate indexes in Fig. 5c using an example from TPC-H [3].

The coordinator always creates a candidate index. Existing tuning frameworks [22, 74, 84] could drop indexes by treating them as discrete actions, with a bit vector entry that indicates existence, or by using workload statistics. Even though Proto-X could use these patches, an agent should instead discern an index's value across all explored configurations and leverage configuration neighborhoods to find opportunities to drop. We defer this as future work.

4 LATENT SPACE CREATION

Proto-X constructs separate smaller latent spaces to exploit the nature of each configuration space with its own estimated benefits and to avoid sampling directly from high-dimensional space. Proto-X stitches these smaller spaces together into a holistic space through proto-actions in Phase II to explore and refine its decision-making. We now discuss the spaces' construction for each action type: (1) knobs, (2) query hints, and (3) discrete (e.g., indexes).

4.1 Knobs

This type covers system, table, and query knobs. Existing DBMSs expose knobs of different types: integral (e.g., buffer pool size), float (e.g., page fetch cost), and boolean (e.g., enable hash joins). Proto-X treats them all as *continuous-type* knobs. Prior techniques observed that similar knob values produce similar observations [35, 59, 85]. For example, a buffer pool size of 1024 MB will have comparable performance to 1025 MB.Proto-X quantizes a knob's value range into equal-width buckets (default is 100 buckets). For knobs with large ranges and non-linear performance (e.g., optimizer cost knobs, memory-related knobs), Proto-X log transforms and quantizes the range to better represent the space.Increasing the quantization trades convergence speed for a finer granularity. Fig. 5a shows how Proto-X obtains each knob's latent space value. Using the knobs' metadata from the DBMS (see Sec. 3.1), Proto-X quantizes and normalizes each knob's values into `[-1,1]` to ensure those knob values are placed nearby in the latent space.

4.2 Query Hints

Fig. 5b illustrates how Proto-X represents query hints. Proto-X first obtains the query hint's metadata (see Sec. 3.1), which includes the allowed values. ❶ It then represents the hint in the latent space as a probability distribution. This representation allows the agent to express preference towards each value, such that ❷ more similar preferences are located nearby in latent space.

We also observe that query hints can interfere with the optimizer's ability to generate optimal plans [59]. For instance, in PostgreSQL, TPC-H [3] Q13 normally executes with parallel scans. However, attaching a *pg_hint_plan* [5] hint to force a parallel scan causes Q13 to execute without parallelism. To address this, Proto-X adds a special "Null" value to each query hint. The agent picks this special value to instruct the coordinator to omit the query hint.

4.3 Discrete

These objects require a specification (e.g., index keys) to create in the DBMS. Proto-X currently supports indexes. We defer more complex discrete types (e.g., materialized views, partial indexes) for future work. Prior approaches represent indexes with encoding schemes [63, 74] (e.g., one-hot). These encodings are decoded into `CREATE INDEX` when deployed. However, these schemes do not guarantee that indexes of comparable performance have nearby representations. Therefore, we construct a learned latent space to enforce this [19, 54]. We illustrate this process in Fig. 5c.

Proto-X uses a custom encoding scheme to support arbitrary indexes. This scheme allows Proto-X to consider orders of magnitude more varied indexes (see Table 1) than prior techniques that use a pre-curated list [74, 84]. ❸ Using the database's metadata (see Sec. 3.1), Proto-X first augments each table's indexable attribute list with a "Null" value. ❹ Proto-X then encodes each index by one-hot encoding the table, followed by one-hot encoding each key.

❺ Proto-X then obtains the index's *benefit score*, which ranks candidates [25, 90] on their potential improvement to the target workload. Proto-X uses the workload's estimated cost as the benefit score and estimates it for read-only OLAP workloads through the query optimizer's `what-if` mechanism [20]. For OLTP workloads, Proto-X employs behavior models [51] since the optimizer does not account for maintenance operations (e.g., index inserts).

With the index representations and associated benefit scores, ❻ Proto-X then trains an encoder-decoder [32] network. The encoder maps representations (e.g., 140-dim for TPC-H) to points in a low-dimensional (e.g., 32-dim for TPC-H) latent space, and the decoder transforms points back to their original representations. To force indexes with similar scores closer, we introduce a *band constraint* [36] that restricts each index's latent space point to a range of values based on its score. This constraint improves Proto-X's efficacy by placing candidates in bands based on their estimated benefit, with more beneficial candidates close to the origin.

❼ Proto-X decodes the raw representation vector from the decoder to obtain a deployable index. Proto-X interprets each index component as a probability distribution and selects the most probable option. Proto-X first decodes the table (e.g., to `LINEITEM`) and then keeps decoding key columns until reaching the "Null" value.

5 HOLON RECOMMENDATION

After building its latent space in Phase I, Proto-X moves on to Phase II. In this phase, Proto-X actively recommends holons to tune the DBMS. We first describe how the agent represents the current DBMS configuration to output a proto-action. We then discuss how the agent selects a holon and conclude with optimizations that assist the agent in reliably finding promising configurations.

5.1 State Representation

In Fig. 4 ❸, Proto-X outputs a proto-action from its actor network based on the current DBMS configuration. There are two choices for how to represent this information:

Telemetry Representation: This approach uses the DBMS's internal performance counters [16] (e.g., pages fetched) that it generates while executing the workload as a substitute for the actual DBMS configuration (e.g., knob values). Prior work commonly employs this representation [70, 84].

Structural Representation: This directly embeds the DBMS's configuration [84]. For example, fields in the representation contain system knobs' values. However, the challenge with this representation is variable-length sets (e.g., existing indexes). Prior work [84] uses a bit vector to encode whether a given object (e.g., index) exists from a pre-determined list, which Proto-X does not use. Concatenating each object's representation is also inadequate, as it requires limiting the set size to ensure a fixed-length representation. Instead, Proto-X constructs a set's representation by obtaining each object's latent space representation and averaging them [81].

Proto-X supports either representation to find promising configurations. The telemetry-based representation is simpler and more readily exploitable by an agent than the structural representation. However, the structural representation is more resilient to the DBMS's background processes, which might distort the collected metrics. We elaborate further in Sec. 7.4.

5.2 Candidate Neighborhood Generation

Using the DBMS's state representation, the agent passes it through its actor network (Fig. 4 ❸) to obtain a proto-action. The agent then breaks the proto-action into *slices* and independently generates a neighborhood for each using similarity. Each slice corresponds to some configuration space (e.g., knobs, query hints, discrete). We next discuss how each space is processed below.

Knobs: Recall from Sec. 4.1 that Proto-X handles all knobs as continuous-type knobs. Based on the knob's metadata, the agent projects the proto-action slice out of the latent space $[-1, 1]$ and quantizes the result to obtain a *neighborhood center*. Consider again PostgreSQL's `shared_buffers` system knob. Assume that this knob only takes on size values (in GB) within the set $\{1, 2, 3, 4, 5\}$ and that its neighborhood center is 3 GB. Then 2 and 4 GB lie within a radius $r = 1$ neighborhood, while 1 and 5 GB have a radius $r = 2$. Using an offset -1, we obtain a similar candidate 2 GB. To generate k candidates within a radius r, the agent generates k integral offsets between $[-r, r]$. The agent shifts the center with each offset to obtain a candidate value. We found a radius of 1–3 and candidate size {10,100} to balance coverage and how quickly the configuration can change. We provide a sensitivity analysis in Sec. 7.5.

Query Hints: Recall from Sec. 4.2 that Proto-X represents query hints in the latent space as a probability distribution. As such, the agent first interprets the proto-action slice as a probability distribution over the value set. The agent then takes the most likely value from the distribution as the *center* and samples the distribution to obtain a neighborhood of k candidates.

Discrete: The agent passes the proto-action slice through the decoder and selects the most probable discrete action (e.g., index) representation as the neighborhood *center*. The agent then applies domain knowledge rules to obtain a neighborhood. For indexes, these *structural rules* generate candidates based on the index's definition (e.g., table, key) and do not imply similar expected performance. We found that rules that guarantee similar performance enable the agent to find better configurations more readily. We illustrate four example rules below.

(*Rule 1*) The first is the *leading prefix* rule. From an example center index of `(a,b,c)`, we obtain candidate indexes `(a,b)` and `(a)` by incrementally relaxing the index's specificity until we reach a single-column index [15]. The DBMS can use each candidate index to satisfy the same queries, albeit with fewer index predicates.

(*Rule 2*) The second is the *index type* rule that controls what data structure to use for an index. By default, Proto-X generates only B+tree indexes. With this rule, Proto-X generates candidate indexes of different types (e.g., hash table, block range index) that share the same key as the center B+tree index.

(*Rule 3*) The third rule targets index `INCLUDE` columns. With this feature, the DBMS stores non-key attributes in an index to increase the likelihood of not having to retrieve tuples for some queries (i.e., covering indexes). Proto-X analyzes the workload to obtain jointly accessed attribute sets and attaches them to applicable indexes. Consider a center index `(a,b)` with co-accessed attribute sets `(a,b,c)` and `(a,d,e)`. Proto-X will then generate two additional candidates: "`(a,b) INCLUDE (c)`" and "`(a,b) INCLUDE (d,e)`".

(*Rule 4*) The fourth rule targets per-index knobs that the DBMS exposes to alter a specific index's behavior. For example, PostgreSQL supports specifying a B+Tree's `fillfactor`, which controls how fully the DBMS packs each B+Tree index page. From each candidate index, Proto-X generates additional candidates with different knob settings specified apriori by the user. Consider a candidate index `(a,b)`. Proto-X will then generate an additional candidate "`(a,b) WITH (fillfactor=100)`".

5.3 Candidate Holon Selection

After the agent generates a neighborhood for each proto-action slice, it combines all the neighborhoods using a Cartesian product to produce a candidate holon set. However, this candidate set may have holons with different performance characteristics. Using each holon's latent space representation, the agent's critic network selects the most promising holon to evaluate.

5.4 Agent Optimizations

Although the agent finds promising configurations, its search process is inherently noisy. Proto-X employs three optimizations to guide exploration with prior experience.

Maximal Query Optimization: Proto-X's tuning space becomes drastically more complex when considering all query options for a workload due to the increased number of query plans, range of performance outcomes, and sparsity of optimal plans. The coordinator assists the agent's decision-making at each step by supplementing the agent's selected query options with query option sets from the DBMS's query optimizer or prior experience. The coordinator then maximizes over those query option sets under the same global configuration (e.g., system knobs, indexes). Proto-X only applies this optimization to OLAP and not OLTP workloads, as the framework assumes that OLAP queries are more complex than OLTP queries (e.g., point-lookup queries) and that OLAP workloads have no dependencies between queries (e.g., `INSERT` followed by `SELECT`). We provide a sensitivity analysis of this in Sec. 7.2.

Exploiting Resets: To balance exploration and exploitation, Proto-X's coordinator periodically resets the DBMS environment after a fixed number of steps. When asked to reset, the coordinator must select a configuration to restore. Restoring the initial configuration maximizes exploration by allowing the agent to explore completely different trajectories. In contrast, restoring to a known configuration (i.e., a *checkpoint*) [33] forces the agent to continue exploiting its neighborhood to discover a better configuration. Proto-X supports resetting to the initial or best-discovered configuration. This tweak allows the agent to build upon prior configurations more readily to discover more complex configurations.

Timeouts: As DBMS tuning tools explore configurations, they often encounter suboptimal ones. Proto-X minimizes the time spent there with two strategies using the application's SLA requirements: (1) query timeout [53, 75] limits a bad query's runtime, and (2) workload timeout [84] limits a suboptimal configuration's time. Proto-X decreases the workload timeout as it discovers better configurations. Based on empirical trials and prior work [75, 84], we utilize query timeouts of 15–30s and workload timeouts of 5–10m.

6 EVALUATION

We evaluate Proto-X's ability to optimize a DBMS's configuration for analytical and transactional workloads. 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 tuning indexes and query options in PostgreSQL, we install the *HypoPG* [4] v1.4 and a patched version of *pg_hint_plan* [5] v1.6 extensions.

We evaluate three OLAP workloads and configure all agents to minimize the overall workload runtime. **JOB** [44] is a benchmark that stresses the query optimizer with 21 tables and 113 queries. **TPC-H SF10** [3] models a business analytics workload with eight tables and 22 queries. **DSB SF10** [24] is Microsoft's extension of TPC-DS [68] 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 [29]. We unsuccessfully attempted to rewrite those queries with Apache Calcite [14] so that PostgreSQL could complete them in a reasonable time.

We also evaluate **TPC-C** [1] SF100 with 40 terminals but modify it to remove all secondary indexes and the unique index on the `OORDER` table. We sample 1m runs through BenchBase [23] and configure agents to maximize throughput.

We first discuss the configurations for Proto-X (Sec. 6.1) and other baselines (Sec. 6.2). We then present our results in Secs. 6.3 to 6.5. We defer sensitivity studies into Proto-X to Sec. 7.

6.1 Proto-X Configuration

We configure Proto-X to tune indexes and global knobs for all the workloads. We allow Proto-X to extract indexable attributes from workload predicates [65] and to generate arbitrary index candidates. We discuss OLAP- and OLTP-specific workload settings below:

OLAP: Proto-X tunes 45 system-wide knobs and tunes indexes using all structural rules from Sec. 5.2. We allow Proto-X to build hash, B+Tree, and block range indexes. For B+Tree indexes, it tunes how full the DBMS packs each page by setting `fillfactor` to 90 (default) or 100 (full). For block range indexes, it tunes the summarization granularity (`pages_per_range`) by setting it to 64, 128 (default), or 256 pages. Proto-X tunes for each query 12 optimizer-related knobs[1], per-table access methods[2], and a query hint instructing which table to scan in parallel[3]. For our maximal query optimization (Sec. 5.4), the coordinator runs supplemental query option sets in the following order: (1) the prior step's, (2) the optimizer's, (3) the agent's, and (4) optional domain knowledge-based sets (e.g., index nested loop join, table scans into hash joins).

OLTP: Proto-X tunes 47 system-wide knobs in PostgreSQL and a `fillfactor` table knob that controls how much space the DBMS leaves in a page for updates. Proto-X does not enable the index type or include rules (Sec. 5.2) for two reasons: (1) PostgreSQL cannot build multi-column hash indexes, and (2) attaching `INCLUDE` columns to an index orchanging its type to prevent point-lookups (e.g., block range index) degrades OLTP performance. Proto-X tunes how full the DBMS packs each B+Tree index page by setting `fillfactor` to 90 (default) or 100 (full). Some PostgreSQL knobs have externalities that agents cannot capture [59, 70]. For instance, increasing `max_wal_size` will improve throughput but also increase recovery time. As such, we set `commit_delay` to 0 to make transactions immediately durable, restrict `max_wal_size` to 16 GB, and do not tune autovacuum knobs due to sampling constraints.

Before tuning, Proto-X creates the latent space for discrete indexes. We sample B+tree indexes without `INCLUDE` columns and aim for reasonable coverage of the space. Using 40 parallel instances and the target workload, we sample each benchmark with the following counts:

- **JOB**: 2048 per-table.
- **TPC-H**: 2048 per-table, except $2^{(16-1)}$ for `LINEITEM`.
- **DSB**: 1024 on small (less than seven indexable attributes) tables. Otherwise, we generate 8192 per (table, attribute).

[1] Query Knobs: `enable_sort`,`enable_memoize`,`enable_hashjoin`,`enable_nestloop`, `enable_mergejoin`,`enable_gathermerge`,`enable_hashagg`,`enable_material`, `enable_parallel_hash`,`random_page_cost`,`seq_page_cost`,`hash_mem_multiplier`
[2] Query Hint: `NoSeqScan(t)`,`SeqScan(t)`
[3] Query Hint: `Parallel(t)`

Table 1: Configuration Space Size – Number of choices considered by each tuning method in our evaluation. "N/A" indicates that the method does not support those options. "Q.Knobs" refers to query knobs, and "Q.Hints" refers to query hints.

	JOB				TPC-H				DSB				TPC-C		
	Knob	Index	Q.Knobs	Q.Hints	Knob	Index	Q.Knobs	Q.Hints	Knob	Index	Q.Knobs	Q.Hints	Knob	Index	Table Knobs
P+DTA-S+A	N/A	73	1356	N/A	N/A	76	264	N/A	N/A	183	588	N/A	N/A	9	N/A
P+DTA-F+A	N/A	201	1356	N/A	N/A	183	264	N/A	N/A	542	588	N/A	N/A	36	N/A
UDO	24	181	N/A	N/A	24	65	N/A	N/A	24	4561	N/A	N/A	33	80	N/A
UniTune	61	59	N/A	N/A	61	53	N/A	N/A	61	263	N/A	N/A	47	91	9
Proto-X	45	2740	1356	1090	45	2^{28}	264	108	45	2^{47}	588	426	47	217	9

- **TPC-C**: 1024 per-table using behavior models [51] to account for maintenance costs from index updates.

We build latent spaces that balance *reconstruction loss* and similarity to the benefit score distribution. The reconstruction loss measures how accurately the decoder reconstructs actions from latent space points. Proto-X estimates distribution similarity by sampling 8192 points from each latent space band, decoding them, and measuring the `(table,key)` distribution. We set all agent parameters (e.g., networks, learning rate) based on prior techniques and existing ML literature [12, 55, 82]. We disclose them here [8].

6.2 Other Tuning Frameworks

Next, we describe the other state-of-the-art automated tuning frameworks we use in our comparison with Proto-X.

PGTune+Dexter (P+D): This baseline first runs PGTune [2], a heuristics-based knob tuner. We then run Dexter [34] to recommend indexes based on HypoPG [4] and PostgreSQL optimizer's workload costs. For TPC-C, we provide a representative query trace to Dexter.

PGTune+DTA-S+AutoSteer (P+DTA-S+A): This runs PGTune, followed by Microsoft's Anytime Database Tuning Advisor (DTA) algorithm [21]. DTA-S uses Hyrise's implementation with their settings [38]: (1) no storage budget, (2) 30m tuning budget, and (3) only two-column indexes. After DTA-S finishes, we run AutoSteer [13], which tunes query knobs by greedily toggling and merging boolean knobs. We configure AutoSteer to tune the same knobs as Proto-X and add toggles to infuse the prior domain knowledge utilized by our maximal query optimization.

PGTune+DTA-F+AutoSteer (P+DTA-F+A): We replace DTA-S in P+DTA-S+A with DTA-F. In DTA-F, we allow DTA to consider more indexes without a time limit. Due to OOM issues and PostgreSQL's inability to optimize queries with 1000s of hypothetical indexes, we limit DTA-F to indexes with three, five, three, and four columns for TPC-H, JOB, DSB, and TPC-C, respectively.

UDO: This is a holistic framework that supports tuning both system knobs and indexes [74]. We use UDO's open-source implementation [6] to construct each benchmark's candidate index list and configure the agent with their default parameters. To ensure a fair comparison, we extended UDO's PostgreSQL knob list to include (1) 24 knobs related to query optimization and resource management and (2) `fillfactor` table knobs for TPC-C. We alter UDO to obtain TPC-C samples through BenchBase [23] and employ 30s, 30s, and 60s query timeouts for JOB, DSB, and TPC-H, respectively. We poll UDO every 15m for the best configuration.

UniTune: Lastly, we compare against Alibaba's UniTune framework [7, 84]. We modified it to support PostgreSQL and obtain samples through BenchBase. We use the same parameters released by the authors. UniTune optimizes the same system knobs as Proto-X, and **UniTune+QOpts** uses the same query options as Proto-X with the same prior domain knowledge. We also enable UniTune's query rewriting via Calcite to match their paper. UniTune only constructs single-column indexes because complex indexes lead to a combinatorial explosion in its one-hot representation. We made two improvements to UniTune. First, we run queries serially and set the target objective to minimize workload runtime. We set UniTune's space budget to a high value (2 TB) for comparative fairness and to allow it to more consistently find promising configurations.

6.3 OLAP Performance Comparison

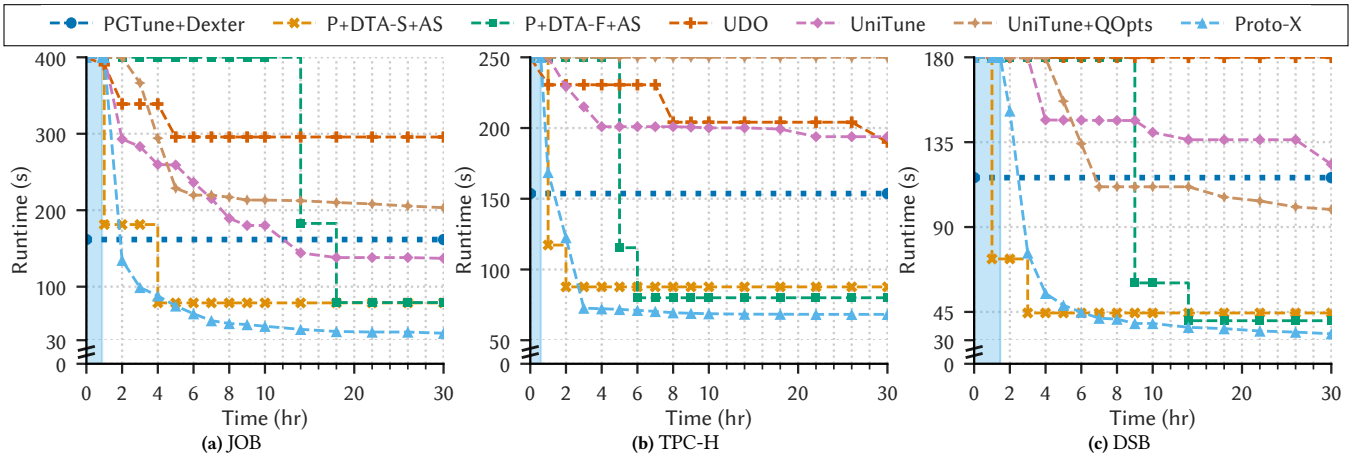
We start by comparing Proto-X to the other frameworks regarding their ability to optimize OLAP workloads. We run the frameworks on the same hardware for multiple *trials* (i.e., independent tuning period) with different random seeds. At the start of each trial, we initialize PostgreSQL with its default configuration and load the database. We then run each agent for 30h to tune the DBMS. As agents utilize timeouts during exploration, we evaluate each discovered configuration without timeouts to obtain their actual performance. After deploying each configuration, we empty the OS page cache, run the workload three times, and report the min [43].

We run four trials of each agent and report results for all trials. In Fig. 6, we plot the mean performance of each agent's best configuration. We also report the best and worst performance achieved by any agent's trial in Table 2. The baseline for all trials is PGTune+Dexter (P+D), as this represents the easiest and fastest method for tuning a DBMS since there is no learning. Our analysis focuses on whether a method generates better configurations than P+D.

JOB: In Fig. 6a, all methods start with the same performance. UDO flattens after 5h and fails to improve over P+D because it does not pick the correct indexes. Although UDO considers 181 index candidates, only 3–5 indexes improve the workload by a measurable amount. Whereas Proto-X's latent space considers the expected benefit of indexes, UDO initially considers all indexes equally. Thus, UDO is unable to reliably pick the correct indexes.

UniTune builds an average of 29 out of 59 index candidates. In Fig. 6a, UniTune surpasses P+D after 12h and flattens after 15h due to the limited tuning options available. UniTune+QOpts has more potential as it tunes query options, but it does not manage the additional complexity and fails to surpass P+D.

Proto-X builds its latent space and surpasses P+D within 2h. After 5h, Proto-X matches P+DTA-S+A and P+DTA-F+A and finds better configurations. Proto-X quickly identifies a high-value set of 2–5 indexes and continues to improve it with additional indexes and query options. We attribute the descent smoothness in part to Proto-X's maximal query optimization (see Sec. 5.4): the agent exploits prior knowledge about promising query option sets while



(a) JOB **(b)** TPC-H **(c)** DSB

Figure 6: OLAP Performance Comparison – The DBMS's performance achieved using the frameworks' configurations over time on JOB, TPC-H, and DSB. We plot the mean performance obtained by four trials of each agent. The shaded region for Proto-X is the time spent constructing the latent space.

Table 2: OLAP Performance Spread –The best and worst performance achieved by a framework's four trials in Fig. 6 on JOB, TPC-H, and DSB.

	JOB		TPC-H		DSB	
	Min	Max	Min	Max	Min	Max
PGTune+Dexter	162s	162s	154s	154s	116s	116s
P+DTA-S+AS	77s	79s	81s	93s	43s	46s
P+DTA-F+AS	78s	80s	72s	87s	40s	40s
UDO	189s	400s	144s	250s	188s	200s
UniTune	132s	146s	176s	200s	117s	125s
UniTune+QOpts	126s	237s	250s	250s	98s	100s
Proto-X	36s	41s	66s	69s	31s	35s

simultaneously exploring unevaluated query option sets, indexes, and global knobs. From Table 2, both Proto-X's worst (41s) and best (36s) trials yield a 46–53% speedup over the next best P+DTA-S+A.

TPC-H: Unlike the previous workload, the results in Fig. 6b show that only Proto-X, P+DTA-S+A, and P+DTA-F+A surpass the baseline P+D for TPC-H. One of this workload's most important actions is to find the high-value `LINEITEM (l_partkey)` index that all frameworks find. However, both UDO and UniTune do not find system knobs better than PGTune. UDO does not correctly tune the number of parallel workers. UniTune dedicates more time to query rewriting and indexes when it should tune system knobs.

P+DTA-S+A and P+DTA-F+A find their best configuration after 2h and 6h, respectively. Proto-X mostly plateaus after 10h, but Proto-X still outperforms P+DTA-S+A and P+DTA-F+A by tuning a query hint that selects a table to scan in parallel. From Table 2, the worst (69s) and best (66s) trial of Proto-X yields a 4–8% speedup over P+DTA-F+A's best run (72s).

DSB: From Fig. 6c, UDO and UniTune do not surpass P+D. Similar to prior workloads, UDO struggles to pick an adequate index set from 4561 candidates, and UniTune does not find promising knob configurations due to misallocating time to index tuning and query rewriting. However, in this case, UniTune+QOpts finds better configurations than P+D by building indexes on `STORE_SALES` and utilizing Proto-X's prior domain knowledge.

P+DTA-S+A and P+DTA-F+A find better configurations after 3h and 14h, respectively, before plateauing. Proto-X spends the first 2h constructing its latent space, surpasses P+DTA-S+A within 6h, maintains its lead over P+DTA-F+A, and continues to find better configurations by building more indexes and tweaking query options. Table 2 shows that Proto-X's worst (35s) and best (31s) trial has a 13–23% speedup over P+DTA-F+A's best run (40s).

6.4 OLTP Performance Comparison

We next compare Proto-X to the other frameworks on their ability to improve TPC-C. Each agent tunes the DBMS for 8h. We report the average of three 1m BenchBase [23] runs for each discovered configuration. We plot the mean performance of the best configuration across the four trials of each agent. We also present each agent's worst, median, and best performance in Table 3.

The results in Fig. 7 show that UDO does not exceed P+D because it does not identify the correct indexes or knob configurations. For OLTP, picking indexes on tables with data actively modified negatively impacts throughput due to additional index maintenance operations. UDO picks indexes that hinder performance, such as indexes on `OORDER` or `ORDER_LINE`.

In comparison, UniTune and Proto-X exceed the performance of P+D by tuning table knobs and avoiding a bad index on `OORDER` that Dexter suggests. From Table 3, Proto-X's best run (13.8k) is better than UniTune's (12.9k) by 7%. Proto-X picks an index on `CUSTOMER` that includes `c_last`. This index improves performance in 47% of TPC-C's transactions (i.e., `Payment`, `OrderStatus`). Proto-X maintains this 7% improvement over P+DTA-S+A (12.9k tps). P+DTA-S+A picks the correct `CUSTOMER` index but also picks bad indexes on `OORDER`. This behavior worsens for P+DTA-F+A (9.1k tps), as DTA considers more (13) but mostly bad indexes.

6.5 Configuration Time Analysis

To better understand Proto-X's tuning behavior, we examine the configurations it generates over time. This analysis shows how Proto-X exploits proto-actions to find impactful configurations despite the high dimensionality of the solution space.

Using Proto-X's best-performing trial on JOB from Sec. 6.3, we track the normalized deviation of configurations over time. We compute deviation as the average absolute difference (e.g., L1-distance) between each configuration and the final configuration's values. For indexes, we track the percentage of indexes that are missing

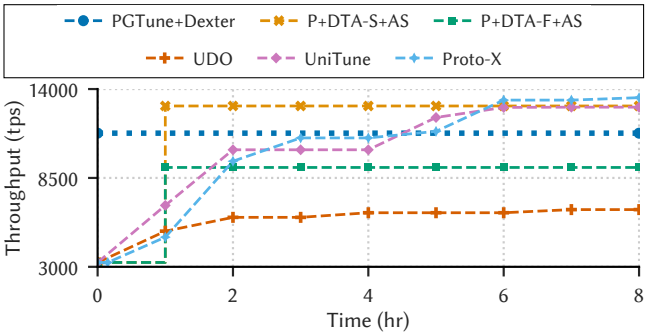
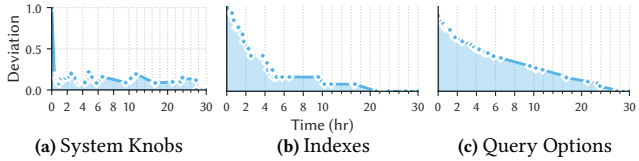


Figure 7: OLTP Performance Comparison – The DBMS's performance achieved using the frameworks' configurations over time on TPC-C. We plot the mean performance obtained by four trials of each agent. The shaded region for Proto-X is the time spent constructing the latent space.

Table 3: OLTP Performance Spread of Frameworks – The worst, median, and best performance achieved by a framework's four trials in Fig. 7.

	TPC-C		
	Min	Median	Max
PGTune+Dexter	11.3k	11.3k	11.3k
P+DTA-S+A	12.9k	12.9k	12.9k
P+DTA-F+A	9.1k	9.1k	9.1k
UDO	5.7k	6.3k	7.6k
UniTune	12.7k	12.8k	12.9k
Proto-X	13.1k	13.4k	13.8k



(a) System Knobs **(b)** Indexes **(c)** Query Options

Figure 8: Configuration Time Analysis – Normalized deviation from the final configuration over time for 12 resource-related system knobs, indexes, and query options using Proto-X's best JOB trial. A configuration's deviation is the average difference of values from the final configuration.

from the final configuration. We only consider 12 resource-related knobs (e.g., workers, memory) for the system knob deviation.

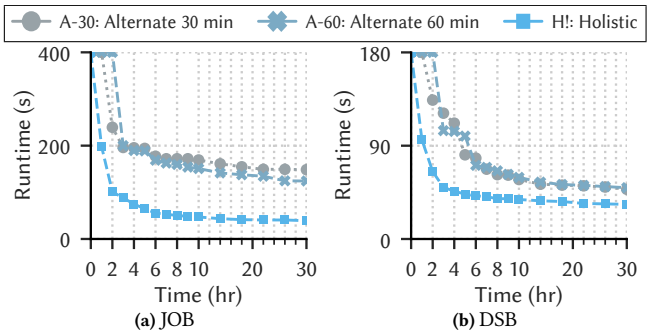
We see in Fig. 8a that the system knobs deviation is about the same after an initial correction in the first hour. Instead of changing them further, the agent focuses on indexes and query options. Fig. 8b shows that Proto-X finds a preliminary set of good indexes after 6h and comes to a final index set after 21h. Contrast this with its behavior in Fig. 8c, where it starts with most of the query options being different and gradually alters them over the course of 30h.

7 SENSITIVITY EXPERIMENTS

We next analyze aspects of Proto-X in more detail. We begin with an ablation study on Proto-X's holistic approach in Sec. 7.1, followed by sensitivity experiments in Secs. 7.2 to 7.6. We conclude with an experiment on generalizing Proto-X's latent space in Sec. 7.7.

7.1 Ablation on Holistic Approach

We first evaluate whether Proto-X outperforms other agents because it holistically reasons across all configuration spaces. We use Proto-X's JOB and DSB configurations from Sec. 6.3 and run three modes: (1) tune each component in 30m intervals (**A-30**), (2) tune each component in 60m intervals (**A-60**), and (3) holistic mode (**H!**).



(a) JOB **(b)** DSB

Figure 9: Ablation on Holistic Approach – DBMS performance using holistic versus sequential tuning of components over time on JOB and DSB. We plot the mean performance obtained by four trials of each mode.

We run each mode four times and plot the mean performance of the best configuration achieved so far over the 30h tuning period in Fig. 9. We discuss each benchmark below.

JOB: In Fig. 9a, H! (39s) finds configurations that are 73.6% and 68.5% better than those found by A-30 (148s) and A-60 (124s), respectively. Both A-30 and A-60 are prone to missing beneficial indexes, with some trials failing to add any indexes at all. We attribute this to the coordination problem: if the query tuner turns off index scans, then the index tuner will not observe an index's benefit. By considering the spaces holistically, H! avoids this problem.

DSB: In Fig. 9b, H! (33s) finds configurations that are 29.8% and 32.7% better than those found by A-30 (47s) and A-60 (49s), respectively. We note that A-30 outperforms A-60 on DSB but performs worse on JOB. Refining this tuning interval may lead to different outcomes. Next, we analyze the composition of each trial's recommended indexes by examining the number of unique one- and two-column indexes. On average, H! builds 13 single-column and 23 two-column indexes, whereas A-30/A-60 builds eight and 11, respectively. By exploring and constructing more diverse indexes, H! finds better configurations than both A-30 and A-60.

7.2 Maximal Query Optimization

The maximal query optimization allows Proto-X to leverage the DBMS optimizer's expertise, past experience, and known priors to guide its recommendations (Sec. 5.4). In this experiment, we consider different modes by combining the following option sets:

- **Agent (A):** Query option set chosen by the agent.
- **Previous (P):** Query option set from the previous time step.
- **Global (G):** Query option set from the query optimizer.
- **Infusion (IV):** Query option sets based on prior domain knowledge about performant query plans to guide the agent. We utilize one option set that enforces index nested loop joins and another that enforces table scans into hash joins.

Using these option sets, we construct the five modes in Table 4 by increasing the amount of guidance. We begin with M1 (**A**), which only uses the agent's chosen query options. We augment M1 with **P** (previous) and **G** (query optimizer expertise) to obtain M2 and M3, respectively. M4 combines both M2 and M3. Lastly, we derive M5 by augmenting M4 with **IV** to infuse prior knowledge about performant query plans. We use the same Proto-X configuration

Table 4: Maximal Query Optimization – Proto-X's worst, mean, and best TPC-H performance under different maximal optimization modes.

Mode	Min	Mean	Max
M1 (**A**)	250s	250s	250s
M2 (**P,A**)	74s	84s	98s
M3 (**G,A**)	91s	93s	94s
M4 (**P,G,A**)	69s	74s	84s
M5 (**P,G,A,IV**)	66s	68s	69s

for tuning TPC-H from Sec. 6.3. As Proto-X's search plateaus in Fig. 6b after 10h, we run each mode for only 10h (four trials each).

Table 4 shows that executing only the agent's query option set (M1) does not find a good configuration. In contrast, M2, M3, and M4 find configurations outperforming the baseline P+D (154s). Some M4 configurations (69s) yield a 4% improvement over P+DTA-F+A (72s) in 8h. Using the **G** and **P** sets, the agent leverages the expert query optimizer and past experience to guide its recommendation of query options (e.g., per-table access methods). This optimization further allows us to imbue the agent with priors from domain knowledge. From Table 4, M5's use of query option sets (**IV**) enables the agent to reduce the best trial's runtime from 69s to 66s and the range of workload runtimes from 15s to 3s.

7.3 Actor-Critic Sensitivity

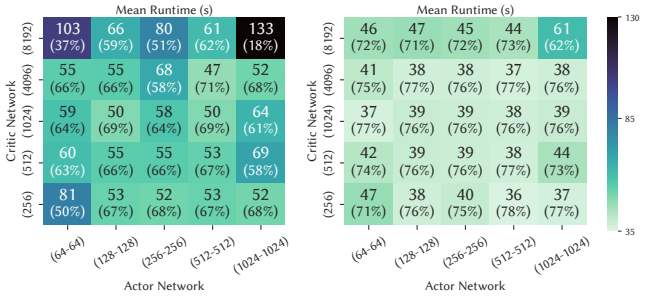
As discussed in Sec. 3, the agent's actor network outputs a proto-action, and the critic network selects the best holon. To understand their impact, we vary five actor and five critic networks of increasing complexity [12, 55, 71, 82]. We run four 30h trials for each pair using Proto-X's JOB configuration from Sec. 6.3. We report the mean performance and percent improvement over P+D in Fig. 10 at 8h and 30h.

From the 8h grid in Fig. 10a, the lower-center network pairs have a similar 64–69% improvement over P+D. However, for small (64-64) and large actor networks (1024-1024) and large critic networks (8192), the performance is more varied, with ranges of 37–66%, 18–68%, and 18–62%, respectively. We attribute these differences primarily to network stochasticity due to limited samples.

By 30h in Fig. 10b, all pairs achieve at least 71% improvement over P+D with a 7% spread, except for (1024-1024, 8192)'s 62%. These results illustrate that the agent's performance tends to worsen for overly simple (e.g., actor 64-64) or complex networks (e.g., critic 8192). Whereas overly simple networks lack reasoning ability, larger networks have more parameters that require more exploration data to learn. Although Proto-X finds promising configurations with a range of actor-critic networks given enough time, Proto-X may benefit from hyperparameter optimization [12] in situations with limited tuning budgets and available idle instances [49].

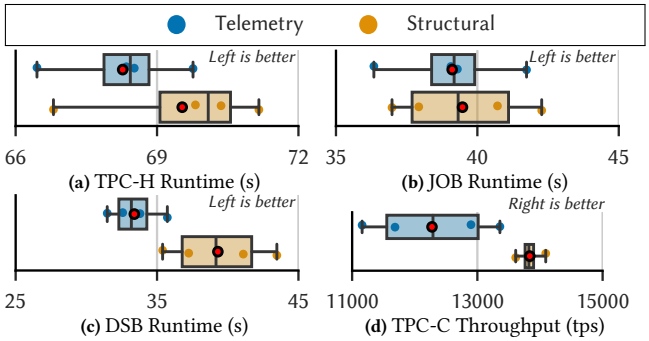
7.4 State Sensitivity

We next analyze the impact of Proto-X's state representation. Recall from Sec. 5.1 that Proto-X utilizes either a telemetry (e.g., workload metrics) or structural (e.g., embed knob values and indexes) representation. Better representations enable the agent to understand the current DBMS configuration and make more targeted recommendations. We run four 30h trials of Proto-X using each representation. In Fig. 11, we report the performance of the best configuration discovered by Proto-X from all trials.



(a) JOB at 8h **(b)** JOB at 30h

Figure 10: Actor-Critic Sensitivity – Proto-X's performance on JOB with different actor-critic network pairs at 8h and 30h. Each cell contains the mean performance and percent improvement over P+D across four runs. Lighter colors correspond to lower workload runtimes.



(a) TPC-H Runtime (s) **(b)** JOB Runtime (s)
(c) DSB Runtime (s) **(d)** TPC-C Throughput (tps)

Figure 11: State Sensitivity – Workload performance after 30h with four runs of each agent using different state representations. The box plot represents min/max and the red dot indicates the sample mean.

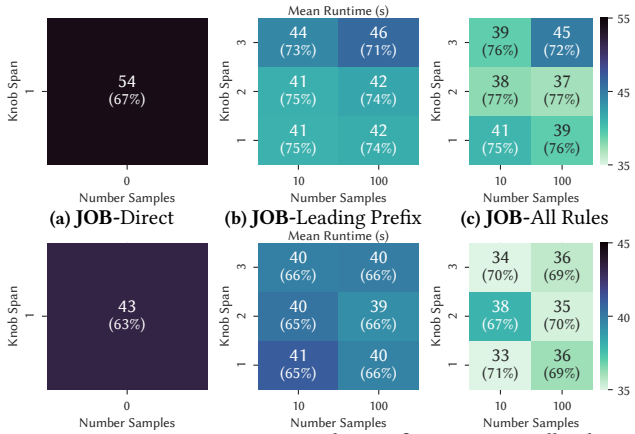
On average, for TPC-H (Fig. 11a) and JOB (Fig. 11b), telemetry performs marginally better than structural by 1s (1.4%) and 0s, respectively. For DSB, Proto-X with telemetry representation finds a configuration 4s (11%) faster than any found by structural, which we attribute to the simpler telemetry-based representation.

Fig. 11d shows that Proto-X's state representation matters the most for TPC-C. Although both representations find good configurations, telemetry has a wider range (2.2k tps) than the structural representation (0.5k tps). When executing OLTP workloads, we do not have fine-grained control over when PostgreSQL runs its background autovacuum worker. Thus, the telemetry representation may be unstable due to these background processes.

7.5 Neighborhood Sensitivity

We next examine Proto-X's neighborhood exploration to find valid candidate holons. To manage combinatoric overhead, Proto-X approximates the neighborhood process with three parameters: (1) knob span or radius (1, 2, 3), (2) number samples (10, 100), (3) the enabled structural rules (leading prefix only, all rules). We introduce a base case that emits the closest valid action (direct), which sets the knob span to 0, number of samples to 1, and disables all structural rules. We run four 30h tuning trials for each configuration using Proto-X's configuration for JOB and DSB from Sec. 6.3. We report the mean performance of each neighborhood and its percent improvement over P+D in Fig. 12.

In Fig. 12, picking the closest valid (direct) action is not the best decision. Constructing a neighborhood allows Proto-X to discover



(a) JOB-Direct **(b) JOB**-Leading Prefix **(c) JOB**-All Rules

(d) DSB-Direct **(e) DSB**-Leading Prefix **(f) DSB**-All Rules

Figure 12: Neighborhood Sensitivity – Performance of Proto-X on JOB and DSB with different neighborhood construction parameters. Each cell contains the mean performance and percent improvement over P+D across four runs. Lighter colors correspond to lower runtimes.

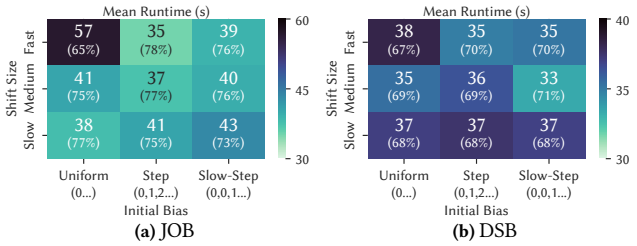
better configurations with further improvements over P+D of up to 10% (JOB) and 8% (DSB). Comparing Fig. 12c to Fig. 12b for JOB and Fig. 12f to Fig. 12e for DSB, enabling more structural rules tends to lead to further performance improvements over P+D: 1–3% (JOB) and 2–7% (DSB). Additional structural rules help facilitate a richer selection of candidate actions (e.g., `INCLUDE` columns, index type). Proto-X benefits from using more structural rules, particularly when they produce candidates with similar or better performance.

For knob span and number of samples in Fig. 12, Proto-X's performance on JOB and DSB slightly degrades as the knob span increases for a given number of samples. Increasing the knob span makes the neighborhood sparser, which increases the agent's stochasticity and hinders materializing beneficial candidates. Although increasing the number of samples can counteract this effect, it makes it more difficult for the critic to differentiate between choices early on. For our experiments, we use a knob span of 1 across TPC-H, DSB, and JOB. We use 10 samples for TPC-H and DSB due to the smaller query knob space and 100 samples for JOB to ensure reasonable neighborhood approximation.

7.6 Exploration Sensitivity

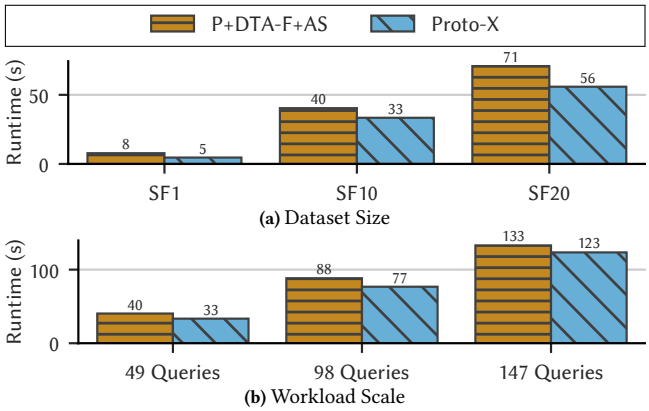
Proto-X augments its proto-action with a bias to encourage exploration (Sec. 3). To measure its impact, we vary the bias's two parameters: (1) initial distribution over the starting episode and (2) shift size at the end of each episode. We choose shift sizes of **slow** (1), **medium** (2), and **fast** (5) to capture a range of speeds and initial distributions of **uniform** (e.g., 0...), **step** (e.g., 0,1,2,...), and **slow-step** (e.g., 0,0,1,...). We run four 30h tuning trials using Proto-X's JOB and DSB configurations from Sec. 6.3. We report the mean performance of each compared to P+D in Fig. 13.

The (Fast, Uniform) bias setting in Fig. 13a stands out, with its mean runtime being >14s worse than all other bias settings. Setting the bias to (Fast, Uniform) prevents the agent from sufficiently covering the latent space, thus leading it to miss important candidate indexes. The remaining bias settings result in similar improvements over P+D, with 73–77% for JOB and 67–71% for DSB. A more targeted bias setting allows the agent to aggressively exploit latent



(a) JOB **(b)** DSB

Figure 13: Exploration Sensitivity – Performance of Proto-X on JOB and DSB with different initial bias settings and increment speeds. Each cell contains the mean performance and percent improvement over P+D in parentheses across four runs. Lighter colors correspond to lower runtimes.



(a) Dataset Size

(b) Workload Scale

Figure 14: Dataset/Workload Drift – Using a latent space built on benefit scores from DSB SF10 with the 49 query workload, Proto-X tunes different scale factors (dataset drift) and workload scales (workload drift). We plot the mean performance of the best configuration discovered from four trials of P+DTA-F+A and Proto-X after 30h.

space regions while ignoring irrelevant ones. We use slower shift sizes in our experiments to ensure sufficient space coverage.

7.7 Generalization

In real-world deployments, the DBMS environment under tuning in Phase II may differ from that in Phase I due to dataset drift, workload drift, or schema changes. Given this, we now analyze whether Proto-X's latent space effectively guides the agent to promising configurations even when its benefit scores become outdated. We first discuss dataset/workload drift, followed by schema changes.

Dataset/Workload Drift: We use DSB for this experiment as it supports dataset scaling (scale factor) and has more complex parameter distributions. We use Proto-X's configuration from Sec. 6.3, with a latent space built on SF10 and the workload's original 49 queries. We simulate dataset drift by evaluating on SF1/SF20 and workload drift by generating 2× and 3× the original workload. We run four 30h trials for each scenario. We use P+DTA-F+A, the next best agent from Sec. 6.3, as the baseline.

From Fig. 14a, Proto-X is better than P+DTA-F+A across all scale factors, with improvements of 38% (SF1), 17.5% (SF10), and 21.1% (SF20), respectively. Indexes that are beneficial at SF10 remain beneficial at SF1/SF20. This continuity allows Proto-X to exploit the SF10 latent space to improve at smaller and larger sizes.

In Fig. 14b, Proto-X uses a latent space built from a smaller workload to tune more complex ones. Proto-X finds better configurations

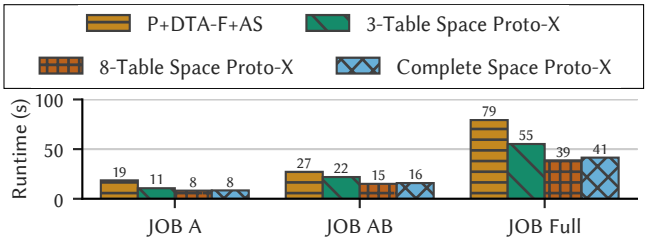


Figure 15: Schema Changes – Mean performance on different JOB workloads by four trials of P+DTA-F+A and Proto-X in 30h. Proto-X trains its latent space based on the workload of JOB AB (66 queries) and considers either 3-tables, 8-tables, or all tables as candidate latent spaces.

than P+DTA-F+A at all workload scales, with improvements of 17.5% (49 queries), 12.5% (98 queries), and 7.5% (147 queries). Although more complex workloads distort the benefit scores of multi-column indexes, Proto-X is hindered more by increased workload runtime, partly due to our maximal query optimization.

Schema Changes: We use JOB to evaluate Proto-X's resilience to schema changes, as its optimal index set touches more tables than the other workloads (Sec. 6.3). We divide the JOB workload into two sets. The first set (**JOB A** / 33 queries) contains all queries in the benchmark whose name ends with "a" (e.g., Q1a). The second set (**JOB AB** / 66 queries) are the queries that end either "a" or "b".

We simulate schema changes by constructing the latent space in Phase I on smaller sets of tables. We build three spaces using the JOB AB workload: 3-Table[4], 8-Table[5], and Complete. We then run four 30h trials and configure Proto-X to tune JOB A, JOB AB, and JOB Full. We present the mean performance of the best-discovered configuration for each scenario in Fig. 15 compared to P+DTA-F+A.

From Fig. 15, Proto-X finds better configurations than P+DTA-F+A in all cases, with improvements of 42–58% (JOB A), 19–44% (JOB AB), and 30–51% (JOB Full). We focus on the differences between the three latent spaces. Fig. 15 shows that 3-Table performs worse due to insufficient coverage of the database's schema (e.g., `MOVIE_COMPANIES`), which causes Proto-X to miss beneficial indexes. Active learning [49] could ensure that the latent space benefit scores remain current as the schema changes.

8-Table and Complete yield roughly similar outcomes in Fig. 15, which indicates the limited benefit of incorporating excess information into the latent space. We note that 8-Table (39s) is competitive, and Complete (41s) performs marginally worse than the latent space built on JOB Full from Fig. 6a (39s). We attribute this to stochasticity in the agent and building the latent space.

8 RELATED WORK

We now discuss existing autonomous DBMS research. Existing research has broadly focused on tuning agents that target a specific configuration space, modeling DBMS internals to facilitate planning decisions, and augmenting DBMS components with ML.

Individual Agents: The literature is rich in individual ML-based tuning agents that broadly cover resource tuning (Bayesian [18, 35, 70] or gradient-based [45, 82]), capacity planning [11, 17, 61], parametric query optimization [27, 69], query tuning (tree search-based [87], model-based [13, 53]), and physical design (partitioning [31, 88, 91], views [10, 80], and indexes [65, 77]).

Joint Configuration Space Tuning: There is limited work on tuning multiple configuration spaces by a single agent. HMAB [60] tunes indexes and materialized views; UDO [74] tunes indexes and knobs, whereas UniTune [84] is capable of tuning indexes, knobs, and queries. HMAB treats the problem as a two-tiered bandit problem where the first tier independently selects views and indexes, and the second selects the combination. UDO and UniTune decompose the entire configuration space and iteratively tune subspaces.

Behavior Models: Modeling aims to produce small and accurate models that can infer the system's performance [48, 51, 55, 63, 89]. Behavior models can substitute for actual DBMS evaluations or prune out parts of the action space [63].

Representations: This work focuses on deriving a query [57, 86] or workload [64, 73] representation that is conducive to downstream tasks (e.g., behavior models, tuning). Creating discriminative representations allow tuning agents to generalize across workloads.

Learned Components: These are traditional DBMS components augmented with machine learning. Existing work has focused on layouts [26], data structures [30, 37], algorithms [40, 62], and query optimization [46, 53, 54, 76, 79]. Other research on learned cardinality estimation has a noticeable impact on the quality of plans the query optimizer generates [42]. Recent work has focused on correctly answering cardinality-related queries with techniques resilient to workload and data changes [56, 67, 72].

9 FUTURE WORK

This work presents the first step towards a holistic or "universal" approach to DBMS optimization. There remain multiple opportunities for further enhancement on top of the core Proto-X framework: (1) online adaptation of the latent space in response to schema changes and drift, (2) incorporating less accurate feedback signals (e.g., partial workload execution), (3) handling user constraints by re-using the agent's exploration trajectories, and (4) improving the agent's introspection abilities to explain its decisions.

10 CONCLUSION

Recent work has focused on combining individual DBMS tuning agents to find better configurations. However, these approaches struggle to coordinate their individual agents. We introduce the framework Proto-X that holistically tunes across multiple configuration spaces. Proto-X organizes the configuration space into a latent space and uses proto-actions to navigate through neighborhoods of similar configurations. Evaluating against other state-of-the-art methods on their ability to tune PostgreSQL for OLAP and OLTP workloads, Proto-X discover configurations that improve up to 53% over the next best approach.

ACKNOWLEDGMENTS

This work was supported (in part) by the CMU Parallel Data Laboratory, VMware Research Grants for Databases, and Google DAPA Research Grants.

[4] `CAST_INFO`, `MOVIE_INFO`, `MOVIE_KEYWORD`
[5] 3-Table, `AKA_NAME`, `MOVIE_COMPANIES`, `MOVIE_INFO_IDX`, `PERSON_INFO`, `TITLE`

REFERENCES

[1] 2010. TPC-C. Retrieved March 2024 from https://www.tpc.org/tpcc

[2] 2022. PGTune – PostgreSQL configuration wizard. Retrieved March 2024 from https://github.com/gregs11094/pgtune

[3] 2022. TPC-H. Retrieved March 2024 from https://www.tpc.org/tpch

[4] 2023. HypoPG. Retrieved March 2024 from https://hypopg.readthedocs.io/

[5] 2023. pg_hint_plan. Retrieved March 2024 from https://github.com/17zhangw/pg_hint_plan/tree/parallel_patch

[6] 2023. UDO. Retrieved March 2024 from https://github.com/jxiw/UDO/

[7] 2023. UniTune. Retrieved March 2024 from https://github.com/Blairruc-pku/UniTune/

[8] 2024. Proto-X Code. Retrieved March 2024 from https://github.com/17zhangw/protox

[9] Sanjay Agrawal, Surajit Chaudhuri, Lubor Kollar, Arun Marathe, Vivek Narasayya, and Manoj Syamala. 2004. Database Tuning Advisor for Microsoft SQL Server 2005. In *VLDB* (vldb ed.). Very Large Data Bases Endowment Inc. https://www.microsoft.com/en-us/research/publication/database-tuning-advisor-for-microsoft-sql-server-2005/

[10] 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

[11] 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

[12] Marcin Andrychowicz, Anton Raichuk, Piotr Stanczyk, Manu Orsini, Sertan Girgin, Raphaël Marinier, Léonard Hussenot, Matthieu Geist, Olivier Pietquin, Marcin Michalski, Sylvain Gelly, and Olivier Bachem. 2021. What Matters for On-Policy Deep Actor-Critic Methods? A Large-Scale Study. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net. https://openreview.net/forum?id=nIAxjsniDzg

[13] 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

[14] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. 2018. Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. 221–230. https://doi.org/10.1145/3183713.3190662

[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] Matthew Butrovich, Wan Shen Lim, Lin Ma, John Rollinson, William Zhang, Yu Xia, and Andrew Pavlo. 2022. Tastes Great! Less Filling! High Performance and Accurate Training Data Collection for Self-Driving Database Management Systems. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 617–630. https://doi.org/10.1145/3514221.3517845

[17] 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

[18] Stefano Cereda, Stefano Valladares, Paolo Cremonesi, and Stefano Doni. 2021. CGPTuner: A Contextual Gaussian Process Bandit Approach for the Automatic Tuning of IT Configurations under Varying Workload Conditions. *Proc. VLDB Endow.* 14, 8 (apr 2021), 1401–1413. https://doi.org/10.14778/3457390.3457404

[19] Yash Chandak, Georgios Theocharous, James Kostas, Scott Jordan, and Philip Thomas. 2019. Learning Action Representations for Reinforcement Learning. In *Proceedings of the 36th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.), Vol. 97. PMLR, 941–950. https://proceedings.mlr.press/v97/chandak19a.html

[20] 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

[21] 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/

[22] Sudipto Das, Miroslav Grbic, Igor Ilic, Isidora Jovandic, Andrija Jovanovic, Vivek R. Narasayya, Miodrag Radulovic, Maja Stikic, Gaoxiang Xu, and Surajit Chaudhuri. 2019. Automatically Indexing Millions of Databases in Microsoft Azure SQL Database. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 666–679. https://doi.org/10.1145/3299869.3314035

[23] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. 2013. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *Proc. VLDB Endow.* 7, 4 (dec 2013), 277–288. https://doi.org/10.14778/2732240.2732246

[24] 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

[25] 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

[26] 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

[27] Lyric Doshi, Vincent Zhuang, Gaurav Jain, Ryan Marcus, Haoyu Huang, Deniz Altinbüken, Eugene Brevdo, and Campbell Fraser. 2023. Kepler: Robust Learning for Parametric Query Optimization. *Proc. ACM Manag. Data* 1, 1, Article 109 (may 2023), 25 pages. https://doi.org/10.1145/3588963

[28] Gabriel Dulac-Arnold, Richard Evans, Hado van Hasselt, Peter Sunehag, Timothy Lillicrap, Jonathan Hunt, Timothy Mann, Theophane Weber, Thomas Degris, and Ben Coppin. 2016. Deep Reinforcement Learning in Large Discrete Action Spaces. arXiv:1512.07679 [cs.AI]

[29] 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*.

[30] 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

[31] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. 2020. DeepDB: Learn from Data, Not from Queries! *Proc. VLDB Endow.* 13, 7 (mar 2020), 992–1005. https://doi.org/10.14778/3384345.3384349

[32] G E Hinton and R R Salakhutdinov. 2006. Reducing the dimensionality of data with neural networks. *Science* 313, 5786 (July 2006), 504–507. https://doi.org/10.1126/science.1127647

[33] Jiechuan Jiang and Zongqing Lu. 2020. Generative Exploration and Exploitation. *Proceedings of the AAAI Conference on Artificial Intelligence* 34, 04 (Apr. 2020), 4337–4344. https://doi.org/10.1609/aaai.v34i04.5858

[34] Andrew Kane. [n.d.]. Introducing Dexter, the Automatic Indexer for Postgres — ankane. Retrieved March 2024 from https://medium.com/@ankane/introducing-dexter-the-automatic-indexer-for-postgres-5f8fa8b28f27

[35] 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/

[36] Mahmut KAYA and Hasan Şakir BİLGE. 2019. Deep Metric Learning: A Survey. *Symmetry* 11, 9 (2019). https://doi.org/10.3390/sym11091066

[37] Eric R. Knorr, Baptiste Lemaire, Andrew Lim, Siqiang Luo, Huanchen Zhang, Stratos Idreos, and Michael Mitzenmacher. 2022. Proteus: A Self-Designing Range Filter. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 1670–1684. https://doi.org/10.1145/3514221.3526167

[38] 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

[39] 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

[40] Ani Kristo, Kapil Vaidya, Ugur Çetintemel, Sanchit Misra, and Tim Kraska. 2020. The Case for a Learned Sorting Algorithm. 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, 1001–1016. https://doi.org/10.1145/3318464.3389752

[41] A. Lawrence. 2002. The Ghost in the Machine. arXiv:astro-ph/0211213 [astro-ph]

[42] 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

[43] Claude Lehmann, Pavel Sulimov, and Kurt Stockinger. 2024. Is Your Learned Query Optimizer Behaving As You Expect? A Machine Learning Perspective. arXiv:2309.01551 [cs.DB]

[44] 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

[45] 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

[46] 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

[47] Wan Shen Lim, Matthew Butrovich, William Zhang, Andrew Crotty, Lin Ma, Peijing Xu, Johannes Gehrke, and Andrew Pavlo. 2023. Database Gyms. In *13th Conference on Innovative Data Systems Research, CIDR 2023, Amsterdam, The Netherlands, January 8-11, 2023*. www.cidrdb.org. https://www.cidrdb.org/cidr2023/papers/p27-lim.pdf

[48] Chenghao Lyu, Qi Fan, Fei Song, Arnab Sinha, Yanlei Diao, Wei Chen, Li Ma, Yihui Feng, Yaliang Li, Kai Zeng, and Jingren Zhou. 2022. Fine-Grained Modeling and Optimization for Intelligent Resource Management in Big Data Processing. *Proc. VLDB Endow.* 15, 11 (jul 2022), 3098–3111. https://doi.org/10.14778/3551793.3551855

[49] 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

[50] 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

[51] 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

[52] Ryan Marcus. 2023. Learned Query Superoptimization. arXiv:2303.15308 [cs.DB]

[53] 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

[54] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A Learned Query Optimizer. *Proc. VLDB Endow.* 12, 11 (jul 2019), 1705–1718. https://doi.org/10.14778/3342263.3342644

[55] Ryan Marcus and Olga Papaemmanouil. 2019. Plan-Structured Deep Neural Network Models for Query Performance Prediction. *Proc. VLDB Endow.* 12, 11 (jul 2019), 1733–1746. https://doi.org/10.14778/3342263.3342646

[56] 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

[57] Debjyoti Paul, Jie Cao, Feifei Li, and Vivek Srikumar. 2021. Database Workload Characterization with Query Plan Encoders. *Proc. VLDB Endow.* 15, 4 (2021), 923–935. https://doi.org/10.14778/3503585.3503600

[58] 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*. https://db.cs.cmu.edu/papers/2017/p42-pavlo-cidr17.pdf

[59] 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

[60] R. Malinga Perera, Bastian Oetomo, Benjamin I. P. Rubinstein, and Renata Borovica-Gajic. 2022. HMAB: Self-Driving Hierarchy of Bandits for Integrated Physical Database Design Tuning. *Proc. VLDB Endow.* 16, 2 (oct 2022), 216–229. https://doi.org/10.14778/3565816.3565824

[61] Olga Poppe, Tayo Amuneke, Dalitso Banda, Aritra De, Ari Green, Manon Knoertzer, Ehi Nosakhare, Karthik Rajendran, Deepak Shankargouda, Meina Wang, Alan Au, Carlo Curino, Qun Guo, Alekh Jindal, Ajay Kalhan, Morgan Oslake, Sonia Parchani, Vijay Ramani, Raj Sellappan, Saikat Sen, Sheetal Shrotri, Soundararajan Srinivasan, Ping Xia, Shize Xu, Alicia Yang, and Yiwen Zhu. 2020. Seagull: An Infrastructure for Load Prediction and Optimized Resource Allocation. *Proc. VLDB Endow.* 14, 2 (oct 2020), 154–162. https://doi.org/10.14778/3425879.3425886

[62] 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

[63] 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

[64] Tarique Siddiqui, Saehan Jo, Wentao Wu, Chi Wang, Vivek Narasayya, and Surajit Chaudhuri. 2022. ISUM: Efficiently Compressing Large and Complex Workloads for Scalable Index Tuning. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 660–673. https://doi.org/10.1145/3514221.3526152

[65] 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

[66] Herbert A. Simon. 1996. *The Sciences of the Artificial* (3 ed.). MIT Press, Cambridge, MA.

[67] Ji Sun, Jintao Zhang, Zhaoyan Sun, Guoliang Li, and Nan Tang. 2021. Learned Cardinality Estimation: A Design Space Exploration and a Comparative Evaluation. *Proc. VLDB Endow.* 15, 1 (sep 2021), 85–97. https://doi.org/10.14778/3485450.3485459

[68] The Transaction Processing Council. 2021. TPC-DS Benchmark (Revision 3.2.0). Retrieved March 2024 from https://www.tpc.org/TPC_Documents_Current_Versions/pdf/TPC-DS_v3.2.0.pdf

[69] Kapil Vaidya, Anshuman Dutt, Vivek R. Narasayya, and Surajit Chaudhuri. 2021. Leveraging Query Logs and Machine Learning for Parametric Query Optimization. *Proc. VLDB Endow.* 15, 3 (2021), 401–413. https://doi.org/10.14778/3494124.3494126

[70] 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

[71] Dana Van Aken, Dongsheng Yang, Sebastien Brillard, Ari Fiorino, Bohan Zhang, Christian Bilien, and Andrew Pavlo. 2021. An Inquiry into Machine Learning-Based Automatic Configuration Tuning Services on Real-World Database Management Systems. *Proc. VLDB Endow.* 14, 7 (mar 2021), 1241–1253. https://doi.org/10.14778/3450980.3450992

[72] 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

[73] 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

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

[75] Junxiong Wang, Immanuel Trummer, and Debabrota Basu. 2021. UDO: Universal Database Optimization using Reinforcement Learning. arXiv:2104.01744 [cs.DB]

[76] Lucas Woltmann, Jerome Thiessat, Claudio Hartmann, Dirk Habich, and Wolfgang Lehner. 2023. FASTgres: Making Learned Query Optimizer Hinting Effective. *Proc. VLDB Endow.* 16, 11 (aug 2023), 3310–3322. https://doi.org/10.14778/3611479.3611528

[77] 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

[78] Ziniu Wu, Ryan Marcus, Zhengchun Liu, Parimarjan Negi, Vikram Nathan, Pascal Pfeil, Gaurav Saxena, Mohammad Rahman, Balakrishnan Narayanaswamy, and Tim Kraska. 2024. Stage: Query Execution Time Prediction in Amazon Redshift.

In *Companion of the 2024 International Conference on Management of Data* (, Santiago AA, Chile,) *(SIGMOD/PODS '24)*. Association for Computing Machinery, New York, NY, USA, 280–294. https://doi.org/10.1145/3626246.3653391

[79] Zongheng Yang, Wei-Lin Chiang, Sifei Luan, Gautam Mittal, Michael Luo, and Ion Stoica. 2022. Balsa: Learning a Query Optimizer Without Expert Demonstrations. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 931–944. https://doi.org/10.1145/3514221.3517885

[80] Haitao Yuan, Guoliang Li, Ling Feng, Ji Sun, and Yue Han. 2020. Automatic View Generation with Deep Learning and Reinforcement Learning. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. 1501–1512. https://doi.org/10.1109/ICDE48307.2020.00133

[81] Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Russ R Salakhutdinov, and Alexander J Smola. 2017. Deep Sets. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2017/file/f22e4747da1aa27e363d86d40ff442fe-Paper.pdf

[82] 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

[83] 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

[84] 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

[85] 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

[86] 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

[87] Xuanhe Zhou, Guoliang Li, Chengliang Chai, and Jianhua Feng. 2021. A Learned Query Rewrite System using Monte Carlo Tree Search. *Proc. VLDB Endow.* 15, 1 (2021), 46–58. https://doi.org/10.14778/3485450.3485456

[88] 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

[89] Xuanhe Zhou, Ji Sun, Guoliang Li, and Jianhua Feng. 2020. Query Performance Prediction for Concurrent Queries Using Graph Embedding. *Proc. VLDB Endow.* 13, 9 (may 2020), 1416–1428. https://doi.org/10.14778/3397230.3397238

[90] Rong Zhu, Wei Chen, Bolin Ding, Xingguang Chen, Andreas Pfadler, Ziniu Wu, and Jingren Zhou. 2023. Lero: A Learning-to-Rank Query Optimizer. *Proc. VLDB Endow.* 16, 6 (apr 2023), 1466–1479. https://doi.org/10.14778/3583140.3583160

[91] Jia Zou, Amitabh Das, Pratik Barhate, Arun Iyengar, Binhang Yuan, Dimitrije Jankov, and Chris Jermaine. 2021. Lachesis: Automatic Partitioning for UDF-Centric Analytics. *Proc. VLDB Endow.* 14, 8 (apr 2021), 1262–1275. https://doi.org/10.14778/3457390.3457392