The VLDB Journal
https://doi.org/10.1007/s00778-023-00831-y

SPECIAL ISSUE PAPER

# DB-BERT: making database tuning tools "read" the manual

**Immanuel Trummer**[1]

Received: 31 January 2023 / Revised: 19 August 2023 / Accepted: 12 November 2023


**Abstract**

DB-BERT is a database tuning tool that exploits information gained via natural language analysis of manuals and other relevant text documents. It uses text to identify database system parameters to tune as well as recommended parameter values. DB-BERT applies large, pre-trained language models (specifically, the BERT model) for text analysis. During an initial training phase, it fine-tunes model weights in order to translate natural language hints into recommended settings. At run time, DB-BERT learns to aggregate, adapt, and prioritize hints to achieve optimal performance for a specific database system and benchmark. Both phases are iterative and use reinforcement learning to guide the selection of tuning settings to evaluate (penalizing settings that the database system rejects while rewarding settings that improve performance). In our experiments, we leverage hundreds of text documents about database tuning as input for DB-BERT. We compare DB-BERT against various baselines, considering different benchmarks (TPC-C and TPC-H), metrics (throughput and run time), as well as database systems (PostgreSQL and MySQL). The experiments demonstrate clearly that DB-BERT benefits from combining general information about database tuning, mined from text documents, with scenario-specific insights, gained via trial runs. The full source code of DB-BERT is available online at https://itrummer.github.io/dbbert/.

**Keywords** Automated database tuning · Text mining · Language models · Reinforcement learning

## 1 Introduction

*Give me a user manual, and I'm happy for hours.*
— — —LennonParham

*When all else fails, read the instructions.*
— — —Anonymous

Manuals are useful. For instance, before starting to tune a database management system (DBMS), it is recommended to read the associated manual. So far, those words of wisdom only seemed to apply to human database administrators. While it is widely acknowledged that database manuals contain useful information, this knowledge has long been considered inaccessible to machines due to barriers in natural language understanding. We believe that this has changed with recent advances in the field of natural language processing, namely by the introduction of powerful, pre-trained language models based on the Transformer architecture [51]. We present DB-BERT, a tuning tool, based on the BERT model [7], that "reads" (i.e., analyzes via natural language tools) the manual and hundreds of text documents with tuning hints in order to find promising settings for database system parameters faster.

The problem of finding optimal values for DBMS parameters (also called "tuning knobs") for specific workloads and performance metrics has received significant attention in recent years. DBMSs often have hundreds of parameters [34], making it very hard to find optimal settings manually. This motivates computational methods for automated parameter tuning. The dominant approach is currently machine learning [1], in particular reinforcement learning [24, 49, 57]. Here, a tuning tool selects value combinations for DBMS parameters to try in a principled manner, guided by the results of benchmark runs for specific settings. However, this approach is expensive (recent work uses hundreds of iterations per tuning session [49]) and works best if guided by input from database experts [17], pre-selecting a small set of parameters to tune and reasonable value ranges to consider. Our goal is to substitute such input by information that is gained automatically by analyzing text documents. We call the corresponding problem variant *Natural Language Processing (NLP)-Enhanced Database Tuning*.

✉ Immanuel Trummer
itrummer@cornell.edu

[1] Cornell University, Ithaca, NY, USA

Published online: 27 December 2023

**Table 1** Example tuning hints with extractions

| Text snippet | Extraction |
|---|---|
| The default value of `shared_buffer` is set very low... The recommended value is 25% of your total machine RAM. [35] | `shared_buffers` $= 0.25 \cdot RAM$ |
| I changed '`random_page_cost`' to 1 and retried the query. This time, PostgreSQL used a Nested Loop and the query finished 50x faster. [33] | `random_page_cost` $= 1$ |
| On a dedicated database server, you might set the buffer pool size to 80% of the machine's physical memory size. [31] | `innodb_buffer_pool_size` $= 0.8 \cdot RAM$ |

DB-BERT extracts, from text, tuning hints that recommend specific values for specific parameters. Instead of focusing on the database manual alone, typically containing recommendations to optimize performance for typical workloads, DB-BERT mines a large number of text documents on the Web. In doing so, DB-BERT is able to access the "long tail" of tuning recommendations, considering less common scenarios as well.

Table 1 shows examples for tuning hints with sources and the associated, formal representation of each extracted hint. Some of the hints (second example) recommend an absolute value while others (first and third example) recommend relative values. For the latter, translating the hint into a concrete value recommendation requires knowledge of system properties such as the amount of RAM. Some of the hints (first two examples) mention the parameter explicitly while others (last example) refer to it only implicitly. DB-BERT can exploit all of the hints shown in Table 1.

For a given text snippet, DB-BERT uses a fine-tuned version of the BERT Transformer model to solve four tasks. First, it decides whether a text snippet contains hints. Second, it translates hints into formulas such as the ones shown in Table 1. This may entail steps for resolving implicit parameter references as well as relative recommendations. Third, instead of relying on hints completely, DB-BERT may decide to deviate from proposed values within pre-defined ranges. Finally, given potentially conflicting hints from multiple sources, DB-BERT chooses weights for hints, representing their relative importance.

DB-BERT does not rely on tuning hints alone. Instead, DB-BERT gains more information via trial runs, executing workloads with specific parameter settings while measuring performance. For instance, this enables DB-BERT to resolve conflicts between recommendations from multiple sources. Trying out recommended values reveals which recommendations are reliable. To decide which values to try, DB-BERT uses reinforcement learning, thereby balancing between exploration and exploitation in a principled manner.

During a tuning session, DB-BERT iterates until a user-defined optimization time budget runs out. In each iteration, DB-BERT selects one or multiple DBMS configurations (i.e., parameter settings) to try out. DB-BERT translates the performance observed during those runs (on user-defined benchmarks) into a reward value. This reward value is used to guide the selection of configurations in future iterations, using the Double Deep Q-Networks [50] reinforcement learning algorithm. To apply this algorithm, we formulate database tuning as a Markov Decision Process (MDP) with discrete states and actions. We represent treatment for each hint as a sequence of decisions, determining the hint type (e.g., relative versus absolute values) as well as the hint weight. To leverage NLP for those decisions, we associate each decision option with a text label. This allows DB-BERT to compare hint text and decision label using the BERT Transformer.

We train DB-BERT in a system and benchmark independent manner, before applying it for specific tuning tasks. In principle, we could use manually annotated tuning documents for training (assigning a high reward for hint translations that are consistent with annotations). However, generating such data requires expert knowledge and is hard to crowdsource (compared to cases where labeling requires only commonsense knowledge [11]). Instead, we exploit the database system itself for (noisy) feedback. We assume that tuning hints, if correctly translated, tend to recommend admissible values that do not to dramatically decrease performance. Hence, we train DB-BERT by assigning rewards for hint translations that result in admissible parameter settings (i.e., the DBMS accepts the setting). On the other side, we assign penalties for translations that result in inadmissible parameter settings (i.e., the DBMS rejects the setting) or settings that decrease performance significantly for a simple example workload. The result of training is a model (i.e., weights for around 110 million parameters of the fine-tuned BERT model) that can be used as starting point for tuning other database systems on other benchmarks.

We also present an alternative version of DB-BERT which does not require any scenario-specific training (i.e., no specialized training for database tuning using text). Instead, this variant exploits out-of-the-box language analysis models, pre-trained on standard benchmarks from the NLP domain. More precisely, it maps the problem of extracting recommendations for specific parameters into a question answering problem, using tuning text as context. Also, it uses zero-shot classifiers to associate relative tuning hints with system resources such as RAM, CPU cores, or disk space.

The idea of leveraging text documents for database tuning has been introduced in a recent vision paper [43], published by the same author as the current one. That paper proposes a simple approach based on supervised learning. The approach is trained via tuning hints that have been manually labeled with hint translations. In contrast to that, DB-BERT uses unlabeled text as input. No manual pre-processing is required on this input text. Choices associated with hint translation steps are annotated with manually provided text labels (15 labels in total). However, those labels are not scenario-dependent and we use the same labels across all experiments (Table 3 shows five out of the 15 labels). The same applies to all other tuning parameters introduced in the following sections. Besides the differences in manual labeling overheads, the prior approach is purely based on input text, does not integrate any performance measurements, and is therefore unable to adapt recommendations to specific benchmarks or metrics. Compared to the initial SIGMOD publication that this paper is based upon [45], this current version expands the original approach by a “zero-shot” variant which does not require task-specific training. This variant is evaluated in the experiments. We discuss differences to prior work in Sect. 2 in more detail.

In our experiments, we compare against the latter work as well as against state-of-the-art methods for database tuning without input text. We exploit large document collections, mined by issuing Google queries with relevant keywords, as text input for DB-BERT. We consider different benchmarks (e.g., TPC-C and TPC-H), metrics (throughput and latency), and database systems (MySQL and PostgreSQL). The experiments demonstrate that DB-BERT benefits significantly from information gained via text analysis. In summary, our original, scientific contributions are the following:

- We introduce multiple variants of DB-BERT, a system that combines natural language text documents and run time feedback of benchmark evaluations to guide database tuning.
- We describe the mechanisms used by DB-BERT to extract, prioritize, translate, aggregate, and evaluate tuning hints.
- We evaluate DB-BERT experimentally and compare against baselines, using multiple benchmarks, metrics, and database systems.

The reminder of this paper is organized as follows. We cover required background in learning and NLP in Sect. 2. Then, in Sect. 3, we introduce our problem model and terminology. We give an overview of DB-BERT in Sect. 4. Then, in Sect. 5, we describe how DB-BERT extracts and prioritizes candidate hints from text documents. We show how DB-BERT translates single hints in Sect. 6 and how it aggregates and evaluates hints in Sect. 7. Next, we present a zero-shot variant of DB-BERT in Sect. 8 which does not require any task-specific training data. In Sect. 9, we report experimental results before we conclude with Sect. 10.

## 2 Background and related work

We discuss technologies that DB-BERT is based upon. Also, we describe prior work addressing similar problems as DB-BERT.

### 2.1 Pre-trained language models

The field of NLP has recently seen significant advances across a range of long-standing problems [53]. These advances have been enabled, in particular, by the emergence of large, pre-trained language models [16], based on the Transformer architecture [51]. Such models address two pain points of prior NLP approaches: lack of task-specific training data and bottlenecks in computational resources for training. Language models are trained, using significant computational resources, on tasks for which training data is readily available in large quantities. For instance, masked language modeling [7] (i.e., predicting masked words in a sentence) can use arbitrary Web text for training. Instead of training new models from scratch for other NLP-related tasks, pre-trained models can be used as a starting point. Pre-trained models can be used either via fine-tuning or via prompting. Using fine-tuning, pre-trained models are trained further on task-specific training data. However, due to the use of pre-training, the number of required training samples and computational overheads are reduced by many orders of magnitude [16]. The latest generation of language models [6, 25, 58], including also OpenAI’s GPT model series [10], can often be used without task-specific training via prompting [4]. Instead of training data, it is sufficient to describe a new task to solve as part of the prompt, the text input to the model.

### 2.2 Applications of language models in data management

Natural language query interfaces [13, 14, 18, 26, 38] are the most popular application of pre-trained models in the context of databases. At the time of writing, corresponding approaches constitute the state of the art for text-to-SQL translation benchmarks (e.g., WikiSQL [59] or SPIDER [56]). The problem of translating text into queries shares certain characteristics with the problem of extracting tuning hints from text. In both cases, text is translated into a formal representation. However, whereas text-to-SQL methods typically translate a single sentence into one single SQL query, DB-BERT extracts multiple tuning hints from multi-sentence text passages. Also, DB-BERT must aggregate and prioritize conflicting hints obtained from multiple sources (a sub-problem that does not appear in the context of natural language query interfaces). Unlike most prior work on text-to-SQL translation, DB-BERT does not assume the presence of labeled training samples.

Recent work explores a variety of novel use cases for large language models in data management [46]. These include applications for data preparation and integration problems [2, 39, 41], data profiling and discovery [5, 19, 47], as well as novel database engine designs that exploit language models for data processing directly [37, 39, 42] or to synthesize code for processing [2, 44].

### 2.3 Reinforcement learning

Reinforcement learning [40] addresses scenarios such as the following. An agent explores an environment, selecting actions based on observations. Those actions may influence the environment (whose dynamics are initially unknown to the agent) and result in reward values. The goal of the agent is to maximize reward, accumulated over time. In order to do so, the agent needs to balance exploration (trying out action sequences about which little is known) with exploitation (exploiting action sequences that seem to work well, based on observations so far). The area of reinforcement learning has produced various algorithms that balance this tradeoff in a principled manner. Specifically, DB-BERT uses the Double Deep Q-Networks [50] algorithm. This algorithm learns to estimate action values in specific states via deep learning, using two separate models for selecting actions and evaluating them.

Reinforcement learning has been used for various problems in the database domain [3, 15, 55, 57], including tuning problems (discussed in detail next). Different from prior work, we combine reinforcement learning with NLP to find promising parameter settings. More broadly, our work connects to prior work on leveraging text for reinforcement learning, in particular prior work on instruction following [27]. However, prior work does not consider performance tuning, specifically database tuning, as we do.

### 2.4 Database tuning

A recent vision paper [43] on NLP-enhanced database tuning, written by the same author as the current publication, relates most to the current work. The prior work trains a Transformer model to recognize sentences containing tuning hints via supervised learning. For sentences classified as tuning hints, it extracts parameters and values according to a simple heuristic. This approach uses only text input but no run time feedback. It extracts a fixed set of recommendations from a document collection, without being able to adapt to specific workloads and performance metrics. DB-BERT, on the other hand, uses hints extracted from text merely as a starting point. It supports a broader range of tuning hints (e.g., implicit hints) and does not require annotated tuning hints during training. We summarize some of the differences in Table 2 and compare both approaches experimentally in Sect. 9.

**Table 2** Comparing DB-BERT to prior work on NLP-enhanced database tuning

| Criterion | Prior-main | This |
|---|---|---|
| Learning type | Supervised | Reinforcement Learning |
| NLP type | Classification | Multiple choice |
| Input | Text | Text + Evaluations |
| Implicit references | No | Yes |
| Adapting hints | No | Yes |
| Iterative | No | Yes |

**Table 3** Labels associated with actions for decision $d = 0$. Placeholders are contained in square brackets

| Action | Label |
|---|---|
| 0 (NO_HINT) | [p] and [v] are unrelated |
| 1 | [p] and [v] relate to main memory |
| 2 | [p] and [v] relate to hard disk |
| 3 | [p] and [v] relate to core counts |
| 4 | Set [p] to [v] |

Machine learning is nowadays the method of choice for many database optimization problems, ranging from query optimization [9, 12, 20, 21, 28, 29, 32, 48] over physical design decisions [8, 15, 23, 55] up to database system parameter tuning [24, 34, 52, 57]. We address an extended version of the latter problem, expanding the input by natural language text documents.

## 3 Problem model

We tune configurations for database system parameters.

**Definition 1** Each DBMS is associated with a set $\mathcal{P}$ of configuration parameters. Denote by $\mathcal{V}$ the set of admissible parameter values. A configuration assigns each parameter to a valid value and is represented as a function $\mathcal{P} \mapsto \mathcal{V}$. Equivalently, we represent this function as set $\{\langle p_i, v_i\rangle\}$ for $p_i \in \mathcal{P}$ and $v_i \in \mathcal{V}$ of parameter-value pairs. Parameters not referenced in a configuration maintain their default values.

Our goal is to find configurations that optimize performance. Traditionally, the following problem model is used.

**Definition 2** A database tuning problem is described by a tuple $\langle b, \mathcal{P}, \mathcal{V} \rangle$. Here, $b$ is a benchmark defining a set of queries (or a transaction workload), together with a performance metric to optimize (e.g., run time or throughput). A solution assigns parameters $\mathcal{P}$, selected for tuning, to values from $\mathcal{V}$ and ideally optimizes performance according to benchmark $b$.

In this work, we address a variant of this problem model.

**Definition 3** An NLP-enhanced database tuning instance is described by a tuple $\langle b, T, S \rangle$. Here, $b$ is a benchmark to optimize and $T$ a collection of text documents containing tuning hints. The goal is to find optimal configurations for $b$, considering all DBMS tuning knobs (more precisely, our current implementation considers all integer, numeric, and Boolean parameters for each system), using tuning hints extracted from $T$ via natural language analysis. $S$ is a vector of numerical system properties (such as the amount of RAM or the number of cores) needed to translate hints, potentially containing relative value suggestions, into concrete values.

We do not expect users to specify parameters to tune nor to suggest value ranges for parameters. We rely on natural language analysis to identify relevant parameters and proposed values. However, the approach presented in this work assumes access to a DBMS instance. Via this interface, we verify whether extracted parameter names are valid and whether the parameter type falls within our scope. Our current implementation considers integer, Boolean, and numeric parameters. This scope covers a large share of performance-relevant parameters in database systems like PostgreSQL and MySQL. An extension to other value types, e.g., string-valued parameters, is, in principle, possible. However, integer, numerical, and Boolean parameters (which can be represented as integers) open up interesting possibilities for NLP-enhanced tuning. For instance, given multiple conflicting value recommendations for the same parameter, it is possible to select a value that minimizes distance to any of the recommendations (which requires a distance function). This is less convenient for non-numerical value types.

The goal of text analysis is to extract tuning hints, described next.

**Definition 4** A tuning hint suggests a value for one DBMS parameter. We model tuning hints as a triple $\langle t, p, v \rangle$ where $t$ is a text snippet containing the hint, $p$ a specific parameter, and $v$ a specific value mentioned in $t$. We call the hint explicit if $p$ is mentioned explicitly in $t$ and implicit otherwise. In pseudo-code, we use notation $h.p$ or $h.t$ to refer to parameter or text of hint $h$.

Note that a text snippet $t$ may contain suggestions for multiple parameters or multiple suggested values for the same parameter. This is why we need $p$ and $v$ to identify a specific hint within $t$. Value $v$ may not always be the concrete value proposed for $p$. This is why we translate tuning hints into formulas, defined next.

**Definition 5** We translate tuning hints $\langle t, p, v \rangle$ into a formula of the form $p = f(v, S)$ where $f$ is a formula and $S$ a vector of numerical system properties (e.g., the amount of main memory). We consider formulas of type $f(v, S) = v \cdot m$ as well as $f(v, S) = v \cdot S_i \cdot m$ where $S_i$ is the $i$-th component of $S$ and $m \in \mathbb{R}$ a multiplicator (picked from a discrete set $M$ of multiplicators).

We illustrate tuning hints and their translation.

***Example 1*** Consider the text snippet $t$ ="Properly configure `shared_buffers` - we recommend 25% of available RAM".[1] Assume $S = \langle 8GB, 4, 1TB \rangle$ describes the amount of RAM, the number of cores, and the amount of disk space on the target system. Then, the tuning hint $\langle t, p, v \rangle$ for $p$ =`shared_buffers` and $v = 0.25$ should translate into the formula $f(v, S) = v \cdot S_0 \cdot 1$ (where 1 represents the multiplicator), which evaluates to 2 GB.

## 4 System overview

Figure 1 shows an overview of the DB-BERT system. DB-BERT searches settings for the tuning knobs of a DBMS that maximize performance according to a specific benchmark (specifying workload and performance metric). DB-BERT differs from prior tuning system in that it exploits text documents about the DBMS to tune, for instance the DBMS manual, as additional input.

DB-BERT obtains as input the benchmark to tune, a collection of text documents containing suggested settings for tuning knobs, and numerical properties describing the hardware platform (namely, our implementation expects the amount of RAM, the number of cores, and the amount of disk space as inputs). The latter input is necessary to translate tuning hints in text documents that use *relative* recommendations (e.g., suggesting a buffer size as a percentage of the amount of RAM). Note that DB-BERT is not restricted to parameters that relate to the aforementioned hardware properties. DB-BERT can process hints for arbitrary parameters, as long as recommended values are specified as *absolute* values in text.

DB-BERT does not use text input alone to determine parameter settings (separating it from prior work on NLP-enhanced database tuning [43]). Instead, it exploits run time

[1] https://blog.timescale.com/blog/13-tips-to-improve-postgresql-insert-performance/.

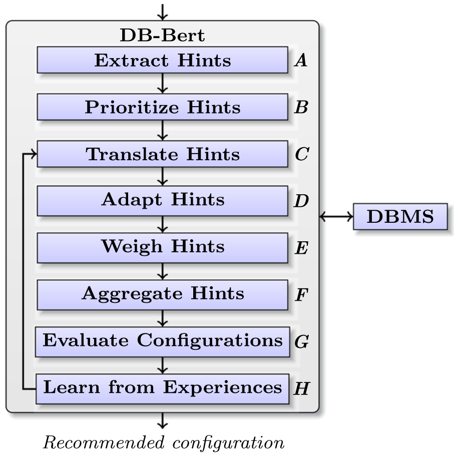


**Fig. 1** Overview of DB-BERT system: we exploit tuning hints, extracted from text documents, to find optimal DBMS knob settings for a given workload

**Algorithm 1** NLP-enhanced database performance tuning.

```
// Optimize all parameters P for benchmark b via hints
// from text collection T, using multiplicators M, system
// properties S, and weights W for translating hints. Use
// ≤ l hints per parameter and episode and up to e hints
// per episode. Evaluate n configurations in each episode.
function DB- BERT(T, b, P, M, S, W, l, e, n)
    // Extract tuning hints from text snippets
    H ← ∪_{t∈T} ExtractHints(P, t)
    // Order tuning hints by priority
    H_o ← OrderHints(H, l)
    while No timeout do
        // Iterate over hints in priority order
        for H_b ← Batches(H_o, e) do
            // Evaluate configurations created using hints
            RunEpisode(b, H, S, M, W, n)
        end for
    end while
    return Best configuration found
end function
```

feedback obtained by benchmarking specific configurations on the DBMS to tune. Hence, DB-BERT requires a connection to a DBMS instance.

At the start of a tuning session, DB-BERT divides input text into text snippets and tries to extract tuning hints from each snippet (Step **A** in Fig. 1). A tuning hint corresponds to a recommendation of a specific value for a specific parameter. Extracting hints from text snippets is non-trivial, in particular as parameter references may be implicit (i.e., the text does not explicitly mention the name of the parameter to tune). Next, DB-BERT determines the order in which hints will be considered in the following stages (Step **B** in Fig. 1). Ideally, the most important hints are considered first. DB-BERT uses a heuristic to order hints, prioritizing hints about frequently mentioned parameters while limiting the number of hints considered consecutively for the same parameter.

Next, DB-BERT iteratively creates configurations (i.e., value assignments for tuning knobs) from tuning hints. It evaluates those configurations on the input benchmark via trial runs. Iterations continue until the user interrupts optimization or a user-specified optimization time limit is reached.

In each iteration, DB-BERT considers a batch of tuning hints (not the entire set of tuning hints). It considers hints in the order established at the start of the tuning session, thereby considering the seemingly most important hints first. For each hint, DB-BERT takes three types of decisions. First, it translates the hint text into a simple equation, assigning a value to a parameter (Step **C** in Fig. 1). Second, in Step **D**, it decides whether to deviate from the recommended value (i.e., whether to multiply the recommended value by a constant). Third, it assigns a weight to the hint (Step **E**). These weights decide how to prioritize in case of conflicting recommendations about the same tuning knob. After treating all hints in the current batch, DB-BERT aggregates them into a small set of configurations (Step **F**), mediating between inconsistent recommendations using hint weights. It evaluates those configurations on the user-specified benchmark via trial runs (Step **G** in Fig. 1).

DB-BERT learns to improve the way hints are translated, adapted, and weighted over the course of a tuning session. This allows DB-BERT to specialize a configuration to the current benchmark and platform. DB-BERT uses reinforcement learning to make all decisions associated with Steps **C** to **E** in Fig. 1. The learning process is therefore driven by a reward function that the system tries to maximize. In case of DB-BERT, that reward function is based on the performance results for specific configurations during trial runs. Configurations that are accepted by the DBMS (i.e., trying to set parameters to specific values does not result in an error) and achieve high performance generate high reward values. Based on rewards received, the system learns to improve its decision making in coming iterations (Step **H** in Fig. 1).

DB-BERT uses *deep* reinforcement learning. This means that immediate and future reward values associated with specific choices are estimated using a neural network. Specifically, DB-BERT uses BERT, a pre-trained language model, as neural network. Due to pre-training, this model comes with powerful natural language analysis capabilities out of the box. To estimate the value of specific choices during Steps **C** to **E**, BERT is applied to pairs of text snippets. The first snippet is taken from the text of a tuning hint, the second snippet is a text label representing the semantics of that choice (see Table 3 in Sect. 6 for example labels). Based on reward values received, the initial weights of the BERT model are refined over the course of a tuning session (in Step **H**).

Algorithm 1 represents the main function, executed by DB-BERT, in pseudo-code. The input integrates user-

**Algorithm 2** Extract candidate tuning hints from text documents.

// Extract tuning hints about parameters $P$ from text $t$.
**function** ExtractHints($P, t$)
  // Extract explicit parameter references
  $E \leftarrow \{p \in P | contains(t, p)\}$
  // Extract implicit parameter references
  $i \leftarrow \arg\min_{p \in P} \delta(BERT(p), BERT(t))$
  // Extract candidate parameter values
  $V \leftarrow$ExtractValues$(t) \cup \{0, 1\}$
  // Return pairs of values and parameters
  **return** $\{\langle t, p, v \rangle | p \in E \cup \{i\}, v \in V\}$
**end function**

**Algorithm 3** Prioritize hints based on their parameters.

// Order hints $H$ using stride of length $l$.
**function** OrderHints($H, l$)
  // Collect parameters in hints
  $P \leftarrow \{h.p | h \in H\}$
  // Group hints by parameter
  $G \leftarrow \{\langle p_i, H_i \rangle | H = \dot{\cup} H_i, h \in H_i \rightarrow h.p = p_i\}$
  // Sort parameters by hint count
  $p_0, \ldots, p_n \leftarrow P$ sorted by number of hints (ascending)
  // Initialize result list
  $R \leftarrow []$
  // Iterate over hint ranges
  **for** $i \leftarrow 0, \ldots, \lceil |G(p_0)/l| \rceil$ **do**
    // Iterate over (ordered) parameters
    **for** $p \leftarrow p_0, \ldots, p_n$ **do**
      // Add hints on $p$ within $i$-th range
      Append$(R, G(p)[i \cdot l : (i + 1) \cdot l - 1])$
    **end for**
  **end for**
  **return** $R$
**end function**

provided inputs, represented in Fig. 1, as well as other parameters, extracted automatically or kept constant across systems and benchmarks. These include the full set of integer, Boolean, and numeric tuning knobs, extracted from the DBMS, $P$, a set $M$ of multiplicators (to deviate from values proposed in text), a set $W$ of weights (to determine relative importance between conflicting hints from different sources), and parameters $l$, $e$, and $n$ to choose the number of hints processed per parameter and iteration, the total number of hints considered per iteration, and the number of configurations evaluated per iteration, respectively. The semantics of those parameters will be described in more detail in the following sections.

Line 8 in Algorithm 1 realizes Step **A** from Fig. 1, Line 10 realizes Step **B**. The main loop iterates until the tuning time budget is depleted. Function Batches($H_o, e$) divides hints into batches of size at most $e$, following the previously established hint order. Each invocation of RunEpisode realizes Steps **C** to **H** from Fig. 1. Finally, DB-BERT recommends the best observed configuration.

Section 5 discusses hint extraction and ordering. Section 6 describes the learning process in more detail and Sect. 7 outlines how hints are aggregated into configurations.

## 5 Extracting candidate hints

In a first step, DB-BERT extracts candidate tuning hints. Following Definition 4, a tuning hint consists of a text snippet, a parameter reference, and a value reference. Algorithm 2 describes the extraction process (illustrated in Fig. 2 as well). It extracts explicit as well as *implicit* parameter references. Implicit references are obtained by comparing the BERT encoding for the text (a vector) against BERT encodings of parameter names, selecting the parameter with minimal cosine distance. We consider all numbers that appear in text, potentially combined with size units, as potential value suggestions. By default, we add values 0 and 1, representing on and off values for Boolean flags, into the set of values (on and off values are often not explicitly mentioned in tuning hints). The set of candidate hints for a given text snippet is the Cartesian product between parameter references and values. This means that our candidates likely contain *erroneous* hints (i.e., parameter-value combinations that are not linked by the text). The task of separating actual from erroneous hints is solved during the translation phase, described in the next section.

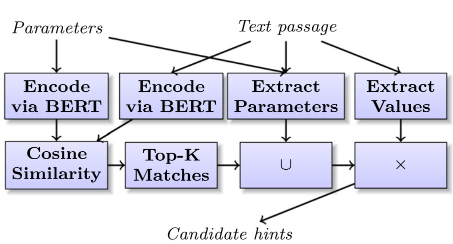


**Fig. 2** Given a text passage and DBMS parameter names, DB-BERT pairs extracted values with parameters that are explicitly mentioned or are similar to the text

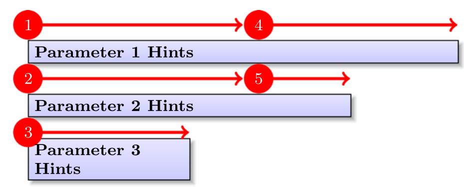


**Fig. 3** DB-BERT prioritizes hints about frequently mentioned parameters while limiting the number of hints per parameters before switching to the next one. In the illustrated example, hints are considered in the order indicated by the red (numbered) arrows

After extracting candidate hints, DB-BERT sorts them using Algorithm 3. Our goal is to increase chances of finding promising configurations when considering hints in sort order. We consider two rules of thumb. First, we expect important parameters to be mentioned in more documents.

**Algorithm 4** Transition function for translating single hints.

```
// For benchmark b, translate hint for parameter p and
// value v, using system properties S, multiplicators M.
// Action a refers to decision d and expands formula f.
// Returns next decision, expanded formula, and reward.
function TSTEP(b, p, v, S, M, d, f, a)
    if d = 0 then
        // Decide hint type and whether to use it
        if a = NO_HINT then
            return ⟨END, −, 0⟩
        else
            return ⟨d + 1, v · S_a, 0⟩
        end if
    else if d = 1 then
        // Choose multiplicator for current hint value
        f ← f · M_a
        // Try setting parameter value and benchmark
        suc ←DBMSSET(p, f)
        if suc = True then
            r ←EVALUATEPERFORMANCE(b)
            return ⟨END, f, r + 1⟩
        else
            return ⟨END, −, −1⟩
        end if
    end if
end function
```

Second, we expect diminishing returns when considering more and more hints about the same parameter. As a result, we prioritize hints about parameters that appear in more documents. However, we consider at most a fixed number of hints about the same parameter, before switching to the next one. Algorithm 3 implements those high-level principles. After grouping hints by parameter, it iterates over hint index ranges. For each index range, it iterates over parameters in decreasing order of occurrences, adding up to $l$ hints per parameter before switching to the next one (until no new hints are left to add for any parameter).

***Example 2*** Fig. 3 illustrates hint ordering with three parameters. Blue rectangles represent hints for each parameter. The horizontal width is proportional to the number of hints. Starting with the most frequently mentioned parameter, we add a limited number of hints for each parameter. After treating the least frequently mentioned parameter (symbolized by the red arrow), Parameter 3, we start again with the first one until no more hints are left.

## 6 Translating single hints

DB-BERT translates tuning hints into arithmetic formulas (see Definition 5 for details). Those formulas may depend on values, specified in text, as well as on system properties such as the amount of main memory. Evaluating a formula yields a value suggestion for a tuning knob.

For each tuning hint, we model the translation as a sequence of decisions. We learn to translate tuning hints by using reinforcement learning. Reinforcement learning is generally applied to Markov Decision Processes (MDPs), specified by a set of states, actions, a transition function mapping state and action pairs to new states, and a reward function. A reinforcement learning agent learns to make decisions maximizing expected rewards, using observations as guidance. In our scenario, states represent (partially specified) arithmetic formulas. Actions specify parts of the formula. The transition functions links partially specified formulas and actions to states representing the formula, completed as specified in the action. The reward function is based on feedback from the DBMS, penalizing translations that result in inadmissible configurations while rewarding changes that improve performance. We describe the structure of the environment (i.e., states, actions, transitions, and rewards) in Sect. 6.1 and the structure of the learning agent in Sect. 6.2.

### 6.1 Learning environment

Algorithm 4 implements the transition function, used by DB-BERT to translate single hints (the pseudo-code is close to the implementation of the step function in the corresponding OpenAI Gym environment[2]). In Algorithm 4, and for a fixed tuning hint, the current state is characterized by a partially specified formula ($f$) and by variable $d$, the integer ID of the next decision to take. For each hint, we start with an empty formula $f$ and $d = 0$. We represent actions (input $a$) as integer numbers from one to five. The semantics of actions depend on the value of $d$. For $d = 0$, the action decides whether the current hint is erroneous (constant NO_HINT) and, if not, whether the hint suggests a relative or absolute parameter value. Relative values are expressed as percentage of system properties such as main memory or the number of cores (stored in vector $S$ with $S_a$ representing a specific vector component). For relative values, we set $f$ to the product between value $v$ and the corresponding system property. We unify treatment of relative and absolute values by setting $S_1 = 1$ (i.e., $a = 1$ represents an absolute value).

For $d = 1$, the action picks a multiplicator from $M$ that allows deviating from the proposed value. Unlike prior work using extracted hints without changes [43], such multiplicators allow DB-BERT to adapt to specific benchmarks. In the next section, we introduce an additional decision that weighs hints. Here, we have fully specified the formula after two decisions. Next, we try setting parameter $p$ to the formula evaluation result. If the setting is rejected by the DBMS, we directly advance to an end state (constant END). This case yields negative reward (motivating our agent to learn

[2] https://gym.openai.com/.

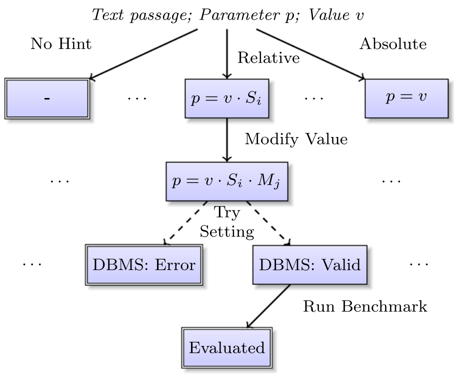


**Fig. 4** Markov Decision Process for hint translation: parameter-value pairs are mapped to formulas by action sequences. Rectangles represent states (double lines mark end states). Arrows represent transitions (dashed arrows mark non-deterministic transitions)

**Algorithm 5** Evaluating expected reward of actions.

// Estimate value of action $a$ for decision $d$,
// given text $t$, parameter $p$ and value $v$.
**function** EVALUATEACTION($t, p, v, d, a$)
  // Get text associated with choice
  $l \leftarrow$ CHOICE_LABEL[$d, a$]
  // Instantiate label for current hint
  $l \leftarrow$INSTANTIATE($l, p, v$)
  // Generate input text for BERT
  $i \leftarrow t \circ l$
  // Generate input types for BERT
  $t \leftarrow 0^{|t|} \circ 1^{|l|}$
  // Generate input mask for BERT
  $u \leftarrow 1^{|i|}$
  **if** MASKED_MODE **then**
    $u \leftarrow$MASK($t, p$)
  **end if**
  **return** BERT($\langle i, t, u \rangle$)
**end function**

translating hints into admissible formulas). Otherwise, we evaluate performance on the input benchmark $b$. The result is a reward value. Higher rewards are associated with better performance. We calculate reward by comparing performance with a configuration to evaluate to performance with default settings. For OLAP benchmarks (e.g., TPC-H), we use the delta of run times (scaled by a constant). For OLTP benchmarks (e.g., TPC-C), we use the throughput delta.

We reward configurations that are admissible and increase performance. Those two metrics are immediately relevant for tuning. We use them when applying DB-BERT for tuning a specific system for a specific benchmark. Before applying DB-BERT for specific tuning tasks, we perform a training phase to fine-tune DB-BERT's language models for hint translation in general. To speed up convergence, only during training, we add an additional component to the reward function. This component rewards settings that seem more likely, e.g., since they are in the same order of magnitude as the default settings for a parameter. Such heuristics replace manually generated hint translations, used in prior work [43]. Figure 4 illustrates the MDP behind the translation process (some of the states in Fig. 4 are not explicitly represented in Algorithm 4).

## 6.2 Learning agent

DB-BERT introduces a learning agent to choose actions in order to maximize rewards. In each state, the agent selects among a discrete set of options. Each option can be expressed as a natural language statement. We can find out which option is correct by comparing that statement against the tuning hint text. Hence, we model action selection as a "multiple choice question answering problem". Pre-trained language models can be used to solve this problem (in our implementation, we use the `BertForMultipleChoice` Transformer model[3]). We fine-tune model weights during training, based on rewards received.

Algorithm 5 shows how the agent evaluates specific actions, based on observations. Besides the action to evaluate, the input includes a description of the current tuning hint (tuning text $t$, parameter $p$, and value $v$) as well as the current translation step (decision $d$). We associate each combination of an action and a decision with a *label*. The array containing those labels is represented via constant CHOICE_LABEL in the pseudo-code. The label is a natural language sentence, representing the semantics of the associated choice. It contains placeholders for the concrete parameter and value in the tuning hint. The INSTANTIATE function replaces placeholders by concrete values.

The BERT model uses three inputs: an input text, a type tag associating input tokens with one of two input types, and a mask indicating tokens to consider. Here, we concatenate hint text and instantiated label to form the input text. Types separate hint text from label. By default, all input text is considered for processing. An exception occurs during our generic training phase (see Sect. 6.1 for more details). Here, we want to avoid learning the names of specific parameters as they do not generalize across systems. Hence, we mask all occurrences of the current parameter name (Function MASK). On the other side, if learning system and benchmark-specific configurations for a concrete tuning problem, there are no reasons to hide information. Algorithm 5 uses a Boolean flag (MASKED_MODE) to switch between these two modes.

Table 3 shows labels associated with different actions and the first decision level. At this level, we decide whether a candidate hint represents an actual hint and, if so, whether the value is relative or absolute. Finally, we illustrate the translation by an example.

[3] https://huggingface.co/transformers/model_doc/bert.html.

**Algorithm 6** Transition function for interpreting multiple hints.

```
1: // Given benchmark b, system properties S, multipli-
2: // cators M, and weights W, translate/weigh hints H.
3: // Evaluate n configurations, based on weighted hints.
4: procedure RunEpisode(b, H, S, M, W, n)
5:     r_e ← 0
6:     H_w ← ∅
7:     // Iterate over batch of hints
8:     for ⟨t, p, v⟩ ∈ H do
9:         // Translate hint into formula
10:        d ← 0
11:        while d ∈ {0, 1} do
12:            a ←ChooseAction(d, p, v, t)
13:            ⟨d, f, r_s⟩ ←TStep(−, p, v, S, M, d, f, a)
14:            r_e ← r_e + r_s
15:        end while
16:        // Add weighted hint if admissible setting
17:        if f ≠ − then
18:            a ←ChooseAction(d, p, v, t)
19:            H_w ← H_w ∪ {⟨W_a, p, f⟩}
20:        end if
21:    end for
22:    // Evaluate weighted hints in combination
23:    r_e ← r_e+EvalWeighted(H_w, b, n)
24:    // Integrate new experiences
25:    UpdateRL(H_w, r_e)
26: end procedure
```

***Example 3*** Consider tuning hint $\langle t, p, v\rangle$ with $t =$"Set `shared_buffers` to 25% of RAM", $v = 25\%$, and finally $p =$`shared_buffers`. First, the agent decides whether the hint is valid and whether it recommends an absolute or relative value. Using the labels from Table 3, the agent evaluates alternative actions based on the hint text. For instance, for action 1, the agent generates the input text "Set `shared_buffers` to 25% of RAM. `shared_buffers` and 25% relate to main memory.", separating the two sentences via the type specification. If masked mode is activated, the two occurrences of the `shared_buffers` parameter are masked. To make a choice, the agent internally compares values resulting from applying BERT to the input for each possible action.

## 7 Aggregating hints

The last section describes how to translate single tuning hints. However, we often need to integrate multiple hints, possibly from different sources, to obtain optimal performance. DB-BERT creates configurations based on groups of hints. This requires aggregating, possibly conflicting hints, from different sources. To support that, we expand the MDP presented in the last section. Instead of considering a single hint, we consider an entire batch of hints. For each single hint, we add an additional decision assigning the hint to a weight. This weight determines the priority when aggregating the hint with others into a configuration. Note that this approach gives DB-BERT the possibility to prioritize hints differently for each input workload. Different from static heuristics for hint priorization, this enables DB-BERT to specialize configurations to each input workload, even when using the same source text.

Algorithm 6 shows complete pseudo-code executed during one iteration of DB-BERT's main loop (Algorithm 6 is invoked by Algorithm 1). From the reinforcement learning perspective, each iteration corresponds to one episode of the associated MDP. Each episode starts from the same starting state, representing the default configuration. The number of hints considered per episode does therefore restrict the maximal number of changes, compared to the default configuration. However, as shown in recent work [17, 49], tuning a small number of tuning knobs is typically sufficient to achieve near-optimal performance.

Algorithm 6 obtains a batch of candidate hints as input. It iterates over those hints and uses Algorithm 4 (Function TStep) to translate single hints (respectively, to determine that a candidate hint is erroneous and should not be considered). We postpone benchmark evaluations by specifying "−" as benchmark parameter for TStep. If successful at translating the current hint into a formula (i.e., $f \neq -$), Algorithm 6 assigns a weight (Line 18). Weights are chosen from a discrete set $W$ of possibilities and are assigned by the learning agent (Function ChooseAction). Finally, the algorithm assembles a set $H_w$ of weighted tuning hints.

Next, we assemble one or several configurations to evaluate, using weighted hints. Algorithm 7 chooses and evaluates configurations, using weighted hints as input. It iterates over parameters mentioned in hints (loop from Line 23–30) and selects a limited number of $n$ values to try ($n$ is a tuning parameter). Values are selected in order to cover the range of suggested values (in hints) as well as possible. We choose values iteratively (loop from Line 26–29). We want to cover values proposed in hints as closely as possible in the following sense. Given a distance function $\delta$ comparing values for the same parameter, our goal is to minimize the maximal, weighted distance between a value proposed in a hint and the closest selected value. Function MaxDist calculates the latter metric, given a weighted set $V$ of values and a set of selected configurations $C$. We select values greedily, minimizing the aforementioned cost function in each step[4]. Note that some tuning knobs can only be set to specific values within their value range (e.g., MySQL's

[4] While this heuristic may seem simplistic, it can be shown that it finds near-optimal solutions. Consider the reduction of MaxDist as a function of selected values in $C$ (fixing $V$ and assigning MaxDist$(\emptyset, V)$ to a large constant). The reduction is sub-modular in the set of selected values, meaning that adding more values shows diminishing returns. As it is also non-negative and monotone (adding values cannot increase the maximal distance), the greedy algorithm corresponds to the algorithm

**Algorithm 7** Evaluate set of weighted tuning hints on benchmark.

1: // Maximal distance from $V$ to nearest value in $C$.
2: **function** MaxDist($C$, $V$)
3:     **return** $\max_{\langle v,w\rangle \in V} w \cdot \min_{c \in C} \delta(v, c)$
4: **end function**
5: // Evaluate configuration $C$ on benchmark $b$.
6: **function** Evaluate($b$, $C$)
7:     $suc \leftarrow$ True
8:     **for** $\langle p, v\rangle \in C$ **do**
9:         $suc \leftarrow suc \wedge$DbmsSet$(p, v)$
10:     **end for**
11:     **if** $suc$ **then**
12:         **return** EvaluatePerformance($b$)
13:     **else**
14:         **return** -1
15:     **end if**
16: **end function**
17: // Evaluate up to $n$ configurations on benchmark $b$,
18: // selecting configurations using weighted hints $H_w$.
19: **procedure** EvalWeighted($H_w$, $b$, $n$)
20:     // Select configurations to cover hints
21:     $P \leftarrow \{p|\langle w, p, v\rangle\}$
22:     $C \leftarrow \{\emptyset\}$
23:     **for** $p \in P$ **do**
24:         $V \leftarrow \{\langle v, w\rangle | w = \sum_{\langle w,p,v\rangle \in H_w} w\}$
25:         $C_p \leftarrow \emptyset$
26:         **for** $i \leftarrow 1, \ldots, n$ **do**
27:             $v^* \leftarrow \arg\min_{v|\langle v,w\rangle \in V}$ MaxDist$(C_p \cup \{v\}, V)$
28:             $C_p \leftarrow C_p \cup \{v^*\}$
29:         **end for**
30:     **end for**
31:     // Compose configurations to evaluate
32:     $C \leftarrow \{\cup_{p \in P} i$-th entry from $C_p | 1 \leq i \leq n\}$
33:     // Evaluate performance of configurations
34:     $E \leftarrow \{$Evaluate$(b, c) | c \in C\}$
35:     **return** $\max_{e \in E} e$
36: **end procedure**

`innodb_buffer_pool_size` must be a multiple of the chunk size [31]). We cannot simply average proposed values.

***Example 4*** Assume we collect hints recommending the following values for parameter `shared_buffers`: 1 GB with weight 1, 2 GB with weight 8, and 8 GB with weight 1. When selecting 1 GB, we obtain maximal weighted distance of $8 \cdot |2 - 1| = 8$ GB from value 2 GB (only distance $1 \cdot |8 - 1| = 7$ GB from 8 GB). Selecting 2 GB yields a maximal weighted distance of 6 GB from value 8 GB. Selecting 8 GB yields a maximal weighted distance of 48 GB from value 2 GB. Hence, we select value 2 GB first. Next, we select value 8 GB to minimize the maximal distance of the remaining values to 1 GB.

Finally, we compose selected values for each parameter into $n$ configurations (Line 32). Function Evaluate evaluates selected configurations on the given benchmark $b$. It assigns a penalty for configurations that are not accepted by the DBMS and, otherwise, calculates reward based on benchmark performance (we use the reward function introduced in Sect. 6.1). Function EvalWeighted returns the maximal reward obtained by any configuration.

**Algorithm 8** Extract tuning hints from text documents, using zero-shot methods.

// Extract tuning hints about parameters $P$ from text $t$.
// Resolve relative hints using system properties $S$.
**function** ExtractHints($P$, $t$, $S$)
    // Extract explicit parameter references
    $E \leftarrow \{p \in P | contains(t, p)\}$
    // Extract implicit parameter references
    $i \leftarrow \arg\min_{p \in P} \delta(BERT(p), BERT(t))$
    // Extract hints for each parameter
    $H \leftarrow \emptyset$
    **for** $p \in E \cup \{i\}$ **do**
        // Question to extract recommendation
        $q \leftarrow$"Which values are recommended for"$\circ p \circ$"?"
        // Obtain answer with confidence
        $\langle a, c\rangle \leftarrow$QA$(t, q)$
        // Check confidence threshold
        **if** $c > \theta$ **then**
            // Extract recommended value
            $v \leftarrow$ExtractValue$(a)$
            // Classify recommendation
            $t \leftarrow$Classify$(a)$
            // Treat relative recommendations
            $v \leftarrow$Update$(v, t, S)$
            // Add tuning hint
            $H \leftarrow H \cup \{\langle p, v\rangle\}$
        **end if**
    **end for**
    **return** $H$
**end function**

## 8 Zero-shot variant

The DB-BERT approach presented so far requires expensive training, specifically for the scenario of database tuning. As new variants of language models appear, this training would have to be repeated. Equally, if targeting different types of tuning text documents, re-training may be necessary for optimal performance. This motivates the "zero-shot" variant presented next. This variant differs from the main version by avoiding scenario-specific training. Instead, it maps the problem of extracting tuning hints into standard problems from the NLP domain such as question answering [36].

### 8.1 Extracting hints

Algorithm 8 shows the algorithm used for analyzing tuning documents. Different from the prior variant of DB-BERT, this algorithm handles hint extraction and hint translation together. This means that reinforcement learning is not

by Nemhauser [30] which guarantees solutions within factor $1 - e^{-1}$ of the optimum.

used anymore to translate text into tuning recommendations. Given a text passage, parameter, and system properties, Algorithm 8 first extracts parameters that relate to the input text. Again, parameter references may be either explicit (i.e., the name of the parameter is explicitly mentioned) or implicit (i.e., the text describes a desired effect while the name of the parameter has to be inferred).

Next, Algorithm 8 iterates over all extracted parameter names. For each parameter, the algorithm generates a question (using a simple template). The purpose of that question is to extract recommended settings for the current parameter. To answer this question, given the tuning text as context, standard methods from the NLP area can be used. The call to sub-function QA represents the invocation of a model for question answering, pre-trained, for instance, on the SQUAD benchmark [36] (or pre-trained on tasks such as text completion which enable the latest generation of language models to solve question answering tasks with a high precision [4]).

Given the lack of task-specific training for database tuning, the chance for spurious extraction may increase. Hence, Algorithm 8 filters answers using a confidence threshold $\theta$, comparing $\theta$ to the confidence score returned for the answer by the question answering model. Assuming that the answer confidence exceeds the threshold, a numerical value is extracted from the recommendation text. This value is either an absolute recommendation or a relative one, referring to system properties such as the amount of RAM, the amount of disk space, or the number of CPU cores. Given a collection of named system properties, the algorithm uses zero-shot classification to compare the recommendation text to a set of text labels (e.g., "RAM", "disk", "cores"). Here, language processing methods typically compare embedding vectors of text labels and a sample to find the class with minimal distance in the embedding vector space. Based on the results of zero-shot classification, the algorithm adapts the extracted value by multiplying with the system property value, associated with the result class.

Finally, the resulting tuning hint is added to the result set, returned by Algorithm 8. Note that tuning hints do not refer to the source text anymore as no further text processing takes place after extraction (different from the prior variant which iteratively translates tuning hints into formulas).

### 8.2 Learning configurations

In this variant of DB-BERT, tuning hints are already translated as part of pre-processing. Hence, the number of decisions made via reinforcement learning reduces. Similarly to before, the algorithm iterates over tuning hints that are sorted using any of the simple heuristic discussed previously. For each of those hints, the reinforcement learning algorithm makes the following decisions:

- Select a multiplicative factor out of a given set of discrete alternatives. This factor adapts the recommended value by going either above or below the raw recommendation.
- Select a weight for the tuning hint. This weight represent the relative importance of the hint. In case of conflicting recommendations, hints with a higher weight are prioritized, as discussed previously.

While the search space for reinforcement learning changes, the reward function (integrating rewards for appropriate value assignments as well as for performance improvements) remains the same. Similarly to before, weighted hints are aggregated into configurations to try, taking into account the weights associated with different hints for the same parameter.

## 9 Experiments

We describe our experimental setup in Sect. 9.1, provide details on the text used for NLP-enhanced database tuning in Sect. 9.2, and details on the training process of all compared algorithms in Sect. 9.3. We compare DB-BERT against various baselines in Sect. 9.4 and study the impact of text document size, data size, and various DB-BERT features on performance in Sect. 9.5.

### 9.1 Experimental setup

We compare approaches for tuning system configuration parameters for MySQL 8.0 and PostgreSQL 13.2. We consider all numerical and Boolean tuning parameters that those systems offer: 232 parameters for PostgreSQL and 266 parameters for MySQL. We use TPC-H with scaling factors one (Sect. 9.4) and ten (Sect. 9.5) and TPC-C with scaling factor 20 as benchmarks. For TPC-C, we use ten terminals, unlimited arrival rate, and 60 s for both, warmup and measurement time (for each trial run). Besides those parameters, we use the default TPC-C configurations for PostgreSQL and MySQL from the OLTP benchmark.[5] In addition to the two TPC benchmarks, we tune for the Join Order Benchmark (JOB) [22]. This benchmark is unusual in that it is designed to challenge the query optimizer in particular. Different from the TPC benchmarks, it has been proposed recently and encountering specialized tuning recommendations for this benchmark on the Web is unlikely. We use all queries of JOB on PostgreSQL and (as time for processing all queries with the default configuration exceeds the intended tuning time frame) the first 20 queries for MySQL. For the analytical benchmarks, each trial run executes all considered

[5] https://github.com/oltpbenchmark/oltpbench.

queries. For each tuning scenario and baseline, we execute five runs and allow for 25 min of tuning time per run (prior work uses the same time frame [57]). All experiments execute on a p3.2xlarge EC2 instance with 8 vCPUs, 61 GB of RAM, and a Tesla V100 GPU featuring 16 GB of memory. The EC2 instance uses the Amazon Deep Learning AMI with Ubuntu 18.04.

We compare against the DDPG++ algorithm [49] as representative for tuning without NLP-enhancement. We consider different value ranges for tuning parameters, ranging from a factor of two around the default value (i.e., $d/2$ to $2 \cdot d$ where $d$ is the default) to 100. In the following plots, we only report results for the factor leading to optimal performance at the end of the tuning period. Also, we compare to two baselines described in a recent vision paper on NLP-enhanced database tuning [43]. In the following, Prior-Main denotes the main method proposed by that prior work, based on supervised learning. Also, we compare against a simple baseline, denoted as Prior-Simple, described in the same paper [43]. Furthermore, we compare to several rule-based tuning tools, specialized for the database management systems we use in our evaluation. For PostgreSQL, we use PgTuner,[6] an online interface that allows users to tune PostgreSQL for specific workload types. In the interface, we provide the precise PostgreSQL version, as well as all relevant hardware properties of the target platform (RAM, disk, and CPUs) as input. For the analytical workloads (TPC-H and JOB), we use tuning recommendations for the data warehouse workload type. For TPC-C, we use recommendations for transactional processing. For MySQL, we use the MySQLTuner.[7] We obtain recommendations by executing the MySQLTuner tool locally on the target platform. In the following plots, legend entry "Specialized" denotes results for the tuning tool (PgTuner or MySQLTuner) matching the tuned database system.

By default, we use the following configuration parameters for DB-BERT. DB-BERT uses reinforcement learning to select multiplicator values and weights for each hint from a fixed set of alternatives. For all experiments, DB-BERT selects the multiplicator from the set $\{1/4, 1/2, 1, 2, 4\}$ and weights from $\{0, 2, 4, 8, 16\}$. We use the same number of alternatives (five) in each case. This makes it easier to model the associated environment with OpenAI's gym framework. We avoid using overly small or large multiplicators (if the optimal parameter value deviates by more than factor four from the proposed value in any direction, the associated hint should be disregarded). The set of weight alternatives allows DB-BERT to disregard hints (by using a weight of zero) as well as to make specific hints up to eight times more important, compared to other hints with non-zero weights. We set $l$ to 10 in order to allow at most ten hints per episode

[6] https://pgtune.leopard.in.ua/#.

[7] https://github.com/major/MySQLTuner-perl.

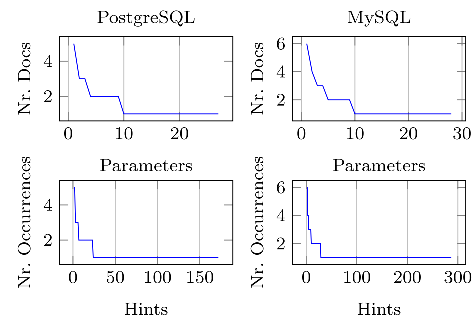


**Fig. 5** Frequency distribution of hints and parameters in the collection of tuning documents for PostgreSQL and MySQL

and parameter. We consider at most 50 hints per episode in total ($e = 50$) and evaluate two configurations per episode ($n = 2$). DB-BERT splits text documents into segments of length at most 128 tokens.

We also evaluate a DB-BERT variant, described in Sect. 8, that parses manuals via a zero-shot approach. This means that no task-specific training is used (unlike for the primary DB-BERT variant). In the following plots, this variant is denoted as DB-BERT0. We use the same settings for all tuning parameters that are shared between DB-BERT and DB-BERT0, except for the parameter determining the number of hints processed per episode. As DB-BERT0 tends to extract less hints than DB-BERT from the same documents, we set the number of hints processed per episode to ten ($e = 10$). For zero-shot classification, we use a BART model, pre-trained on the MNLI benchmark.[8] For question answering, we use a Roberta model, pre-trained on the SQUAD benchmark.[9] For the confidence threshold, we use $\theta = 0.05$.

All baselines (with the exception of specialized tuning tools) are implemented in Python 3.7, using Pytorch 1.8.1 and (for the NLP-enhanced tuning baselines) the Huggingface Transformers library [54]. DB-BERT uses Google's programmable search engine API[10] to retrieve text documents. Also, DB-BERT uses the Double Deep Q-Networks [50] implementation from the Autonomous Learning Library[11] as reinforcement learning algorithm.

## 9.2 Tuning text documents

DB-BERT comes with a script that retrieves text documents via Google search and transforms them into the input format required by DB-BERT. For most of the following experiments, we use two document collections retrieved via the queries "Postgresql performance tuning hints" (issued

[8] https://huggingface.co/facebook/bart-large-mnli.

[9] https://huggingface.co/deepset/roberta-base-squad2.

[10] https://developers.google.com/custom-search.

[11] https://github.com/cpnota/autonomous-learning-library.

**Table 4** Tuning parameters mentioned in most documents for PostgreSQL and MySQL

| System | Parameter |
|---|---|
| PostgreSQL | shared_buffers |
| | max_connections |
| | max_parallel_workers_per_gather |
| | max_wal_size |
| | wal_buffers |
| MySQL | innodb_buffer_pool_size |
| | join_buffer_size |
| | innodb_buffer_pool_instances |
| | max_connections |
| | innodb_flush_log_at_trx_commit |

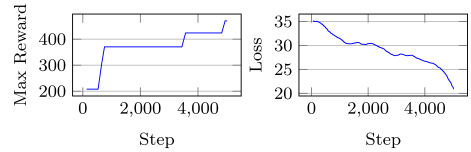


**Fig. 6** Reward and loss when training DB-BERT on Pg100

on April 11, 2021) and "MySQL performance tuning hints" (issued on April 15, 2021). We included the first 100 Google results for each of the two queries into the corresponding document collection (accounting for a total of 1.3 MB of text for PostgreSQL and 2.4 MB of text for MySQL). The results are diverse and cover blog entries, forum discussions (e.g., on Database Administrators Stack Exchange[12]), as well as the online manuals from both database systems. We call the document collection for PostgreSQL Pg100 and the one for MySQL Ms100 in the following.

Figure 5 shows the distribution of parameter mentions and proposed value assignments in those document collections, generated via DB-BERT's candidate hint extraction mechanism (see Sect. 5). Clearly, the distribution of hints over documents and parameters is non-uniform. For both database systems, few parameters are mentioned in multiple documents while most parameters are mentioned only in a single document. Similarly, there are a few assignments proposed by multiple sources. On the other side, most value assignments are proposed only once.

Table 4 shows the most frequently mentioned parameters for both PostgreSQL and MySQL. Parameters related to buffer size (e.g., shared_buffers for PostgreSQL and innodb_buffer_pool_size for MySQL) feature prominently among them. Besides that, parameters related to the degree of parallelism (e.g., the parameter max_parallel_workers_per_gather) or logging (e.g., max_wal_size) are mentioned frequently as well.

### 9.3 Training

Two of the compared algorithms, namely DB-BERT and Prior-Main, use training before run time. Prior-Main uses natural language tuning hints, annotated with associated formulas, as training data. We use the same training samples and training parameters as in the prior work [43]. Consistent with the experimental setup in the latter paper, we apply Prior-Main, trained on PostgreSQL samples, to tune MySQL and Prior-Main, trained on MySQL samples, to tune PostgreSQL. The goal is to demonstrate that NLP-enhanced database tuning does not require system-specific, annotated samples.

Prior-Main does not support extracting benchmark-specific tuning hints from a fixed document collection, a disadvantage if the same document collection is used for tuning multiple benchmarks. To allow at least some degree of variability, we train the Prior-Main model separately for each of our five benchmark runs. This leads to slightly different extractions in each run. Training Prior-Main on the platform outlined in Sect. 9.1 took 417 s for MySQL samples and 393 for PostgreSQL samples.

DB-BERT does not use annotated tuning hints for training. Instead, it uses the database system itself for run time feedback during the training phase. Similar to Prior-Main, we train DB-BERT on Pg100 to tune MySQL and on Ms100 to tune PostgreSQL. We activate the masked mode during training (see Sect. 6), meaning that parameter names are masked. This avoids learning system-specific parameter names (which are useless in our experimental setup) and focuses attention on the sentence structure of tuning hints instead. The reward signal of DB-BERT (see Sects. 6 and 7) combines reward for successfully changing parameter values according to tuning hints (meaning that the corresponding values are valid) and for performance obtained. To measure performance, we use a synthetic database containing two tables with two columns containing consecutive numbers from 1 to 1,000,000. We use a simple count aggregation query joining both tables with an equality predicate. Reward for performance is scaled down by a factor of 100 to avoid specialization to this artificial benchmark (it merely serves to penalize particularly bad configurations such as setting the buffer pool size to a minimal value). Finally, we add a small reward bonus for setting parameter values that are within the same order of magnitude as the default setting (assuming that extreme deviations from default values are possible but less likely). DB-BERT's training starts from the BERT base model [7] with 110 million parameters. All model parameters are tuned during training.

We trained DB-BERT for 5000 iterations on Pg100 and for 10,000 iterations on Ms100 (due to the larger number of hints in this collection). Training took 43 min for Pg100 and

[12] https://dba.stackexchange.com/.

84 min for Ms100. Figure 6 shows progress for Pg100 as a function of the number of training steps.

Note that, in contrast to the primary variant, DB-BERT0 does not require any training.

### 9.4 Comparison with baselines

We compare DB-BERT against baselines on TPC-H (see Fig. 7), JOB (see Fig. 8 with a logarithmic y-axis), and TPC-C (see Fig. 9). We tune PostgreSQL and MySQL for 25 min per run. We use throughput as optimization metric for TPC-C and execution time for TPC-H. We show performance of the best configuration found (y-axis) as a function of optimization time (x-axis). In these and the following plots, we report the arithmetic average as well as the 20th and 80th percentile of five runs (using error bars to show percentiles). The plots contain one data point per baseline for every 30 s of tuning time (not displaying any data points before results for the first trial run have become available for the corresponding baseline). This means that data points in the plots do not necessarily align with the start and end of trial runs.

DDPG++ [49] is a database tuning approach, based on reinforcement learning. It was shown to be competitive with various other state-of-the-art tuning approaches [49]. However, the prior publication evaluates DDPG++ for a few tens of tuning parameters and allocates 150 iterations per tuning session. Here, we consider hundreds of parameters for tuning and aim at a tuning time frame that allows only few iterations. Clearly, within the allocated time frame, DDPG++ does not find solutions of comparable quality to DB-BERT. In particular for TPC-H, DDPG++ often tries parameter changes that decrease performance significantly (e.g., changes to optimizer cost constants triggering different join orders). Hence, performance of the best configuration found remains almost constant for DDPG++ (close to the one achieved via the default configuration, tried before the first iteration). DDPG++ could benefit from specifying parameter-specific value ranges to consider during tuning. For instance, increasing buffer pool size by an order of magnitude, compared to the default settings, is often beneficial. For optimizer cost constants (e.g., `random_page_cost` in PostgreSQL), doing so is however dangerous. Our goal is to show that such input can be partially substituted by information mined automatically from text.

Specializing tuning tools to specific database systems and workload types is another option to avoid costly exploration. We compare to two tuning tools (MySQLTuner and PgTuner) that are specialized to the tuned database systems. Those tuning tools use hard-coded rules to map properties of the target platform and workload to recommended settings. A first advantage of those tools is that they do not require any trial runs with default configuration (hence, they minimize tuning time among all compared tuning tools). Overall, they achieve excellent performance on TPC-H and TPC-C. This is to be expected as those are two of the most popular benchmarks for database management systems. Hence, it can be assumed that the rules used by these tools are optimized to work well for such standard benchmarks. On the other hand, run times on JOB are higher than the optimum by a multiple. This shows that hard-coded tuning rules have limitations when applied to non-standard tuning scenarios.

Prior-Simple and Prior-Main are the two most related baselines as both use tuning text as input, similar to DB-BERT. Prior-Simple uses a naive heuristic for translation. Applying this heuristic is fast and Prior-Simple is typically the first baseline to return results. However, it only extracts the recommendation to set `checkpoint_completion_target` to 0.9 in Pg100 and no recommendations in Ms100. Hence, it does not improve over the default configuration. Prior-Main performs significantly better. Due to small differences in training, extractions differ across different runs, leading to high variance. For instance, for Pg100, Prior-Main is able to extract a tuning hint that recommends setting `shared_buffers` to 25% of main memory in two out of five runs. This can lead to significant performance improvements, in particular for TPC-H. However, average performance is significantly below the optimum. As Prior-Main classifies all sentences in the document collection before aggregating tuning hints, its run time is significantly higher than the one of Prior-Simple.

Both DB-BERT variants find attractive tradeoffs between tuning time and result quality, comparing to generic (i.e., non-specialized) tuning tools. For instance, when tuning for TPC-H, DB-BERT finds settings that lead to significant performance advantages after less than 200 (PostgreSQL), respectively, less than 400 s (MySQL). More precisely, at that point, both DB-BERT variants find settings that increase main memory allocation (e.g., PostgreSQL's `shared_buffers` parameter) or increase the degree of parallellism (e.g., PostgreSQL's `max_parallel_workers_per_gather` parameter), motivated by hints extracted from text, leading to a significant drop in execution time.

Unlike DDPG++, DB-BERT uses tuning text as input that allows identifying the most relevant parameters and candidate values quickly. Compared to Prior-Simple and Prior-Main, it finds significantly better solutions in average. In particular for MySQL, Prior-Main typically fails to find solutions of comparable quality. Furthermore, the time taken by Prior-Main to analyze all documents is typically higher by a factor of two to three, compared to the time until DB-BERT produces a near-optimal solution (i.e., within one percent of DB-BERT's final optimum). Tables 5 and 6 show configurations found by DB-BERT when tuning PostgreSQL. Despite extracting hints from the same document collection, DB-BERT is able to find benchmark-specific configurations.

Compared to system-specific tuning tools, the DB-BERT variants, in particular DB-BERT0, shine on JOB. Hard-coded tuning rules are not flexible enough to optimize performance beyond standard benchmarks. For instance, in the default configuration, PostgreSQL is hampered by sub-optimal choices made by its query optimizer. This is to be expected as the data of JOB is skewed, invalidating assumptions made by the optimizer while estimating cost of candidate plans (e.g., assuming independent predicates). DB-BERT and DB-BERT0 both extract tuning recommendations about PostgreSQL's `effective_cache_size` parameter from text. This parameter represents assumptions of the PostgreSQL planner on the size of the disk cache, available for each query. Both DB-BERT variants set this parameter to a value of 64 for trial runs, following recommendations in text. This value is significantly below the default of 524288. The new setting discourages index scans and leads to different query plans. Further analysis shows that changing the setting for this parameter alone reduces execution time approximately by factor two. Therefore, changing this parameter setting seems to have a similar effect as disabling nested loop joins, a change recommended in the original paper introducing JOB [22]. Unlike recommendations to, e.g., increase buffer pool size, hints concerning the `effective_cache_size` parameter are relatively rare. Enabling DB-BERT to access the "long tail" of tuning recommendations by parsing a large collection of text documents pays off in this case.

In most cases, DB-BERT0 achieves similar performance to DB-BERT, despite the lack of a task-specific training phase. When tuning JOB on MySQL, DB-BERT0 even achieves significantly better results. An analysis of the logs shows that DB-BERT takes longer to find interesting parameter settings due to slightly more noisy extractions, causing DB-BERT to waste the initial trial runs with highly sub-optimal parameter settings that do not appear in text. As JOB requires most time per trial run, this delay prevents DB-BERT from trying efficient settings within the tuning time frame. A follow-up analysis shows that DB-BERT finds configurations with comparable performance to DB-BERT0 after around 2,000 s of tuning time. On the other hand, DB-BERT achieves better results when tuning MySQL for TPC-C. Here, DB-BERT0 consistently converges to a configuration that increases the amount of buffer space (`innodb_buffer_pool_size`) to 1 GB, while using default settings for other parameters. In contrast to that, DB-BERT changes settings for multiple parameters (e.g., parameter `max_connections` and parameter `innodb_flush_log_at_trx_commit`) for significantly higher throughput.

Altogether, DB-BERT0 is the most robust optimization tool across different systems and benchmarks. More precisely, the relative performance degradation of DB-BERT0

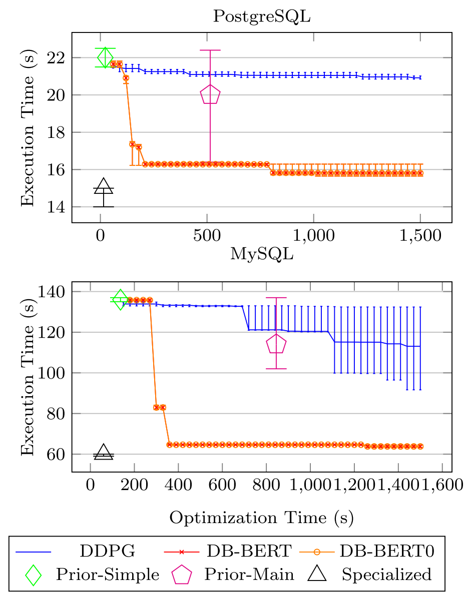


**Fig. 7** Minimal execution time for TPC-H as a function of optimization time for different baselines

(i.e., the relative increase of run time, compared to the optimum, or relative decrease in throughput, compared to the optimum) never exceeds 34% across all scenarios. The degradation reaches its maximum for TPC-C on MySQL where DB-BERT0 only achieves 66% of the optimal throughput. However, all other baselines experience performance degradations of at least up to 93%, compared to the optimum (e.g., when optimizing JOB on MySQL). System-specific tuning tools work well for standard benchmarks but lack the ability to adapt to less common tuning scenarios. Exploiting both, information gained from text documents as well as information gathered via trial runs, gives DB-BERT advantages over methods that exploit only one of those two sources of information. DB-BERT performs similarly to DB-BERT0 in most cases but is slowed down by noisy text extractions in one of the tuning scenarios.

The best configurations for each benchmark and system, considering all runs and all DB-BERT variants, are reported online.[13]

## 9.5 Further analysis

We study the impact of different factors on tuning performance. First, we compare DB-BERT against two simplified variants in Fig. 10. We compare against a variant of DB-BERT that processes hints in document order (instead of prioritizing them as described in Sect. 5). Also, we compare against a variant that does not consider implicit hints (i.e.,

[13] https://drive.google.com/drive/folders/1A_1uvjXzCSrXIoyBh4u46LGmZzukyjjH?usp=sharing.

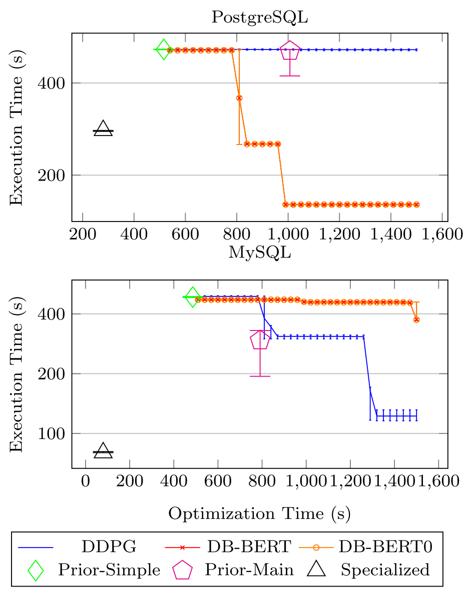


**Fig. 8** Minimal execution time for JOB as a function of optimization time for different baselines

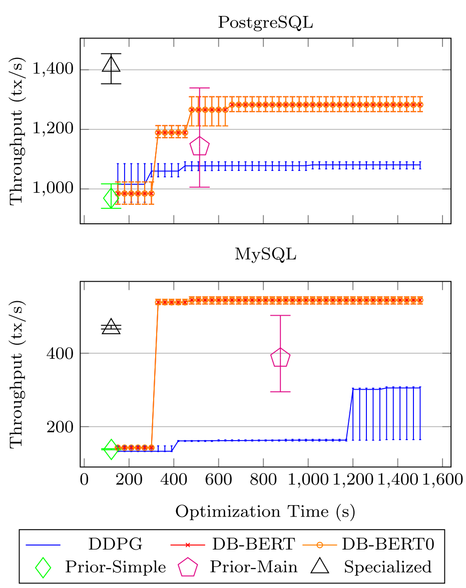


**Fig. 9** Maximal throughput for TPC-C as a function of optimization time for different baselines

**Table 5** PostgreSQL configuration for TPC-H by DB-BERT

| Parameter | Value |
|---|---|
| max_connections | 1100 |
| max_parallel_workers_per_gather | 19 |
| max_wal_size | 4 GB |
| shared_buffers | 1 GB |

**Table 6** PostgreSQL configuration for TPC-C by DB-BERT

| Parameter | Value |
|---|---|
| archive_command | 3 |
| archive_timeout | 4 |
| checkpoint_flush_after | 0 |
| maintenance_work_mem | 32 MB |
| max_wal_senders | 5 |
| random_page_cost | 2 |
| synchronous_commit | 0 |

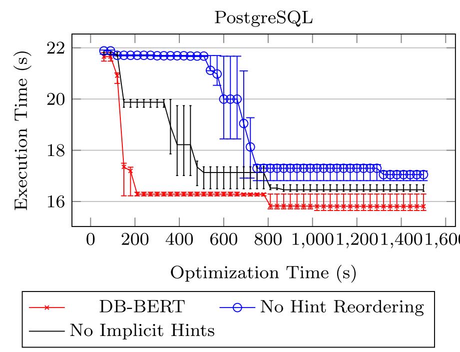


**Fig. 10** Comparison of different DB-BERT variants when optimizing PostgreSQL for TPC-H

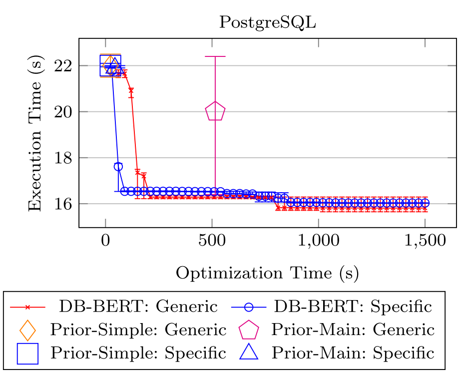


**Fig. 11** NLP-enhanced database tuning for TPC-H on PostgreSQL with different input text (100 documents with generic hints versus one document with benchmark-specific hints)

only hints where parameter names are explicitly mentioned). Clearly, both simplifications degrade tuning performance on TPC-H. Considering hints in document order prevents DB-BERT from tuning the most relevant parameters first. Discarding implicit hints reduces the total set of available hints.

Next, we study the impact of the input text. We replace Pg100, containing hundreds of generic tuning hints, by a single blog post.[14] This post describes how to tune PostgreSQL

[14] http://rhaas.blogspot.com/2016/04/postgresql-96-with-parallel-query-vs.html.

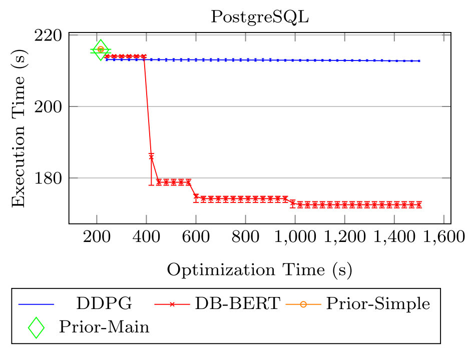


**Fig. 12** Minimal execution time for TPC-H with scaling factor 10 as a function of optimization time

specifically for TPC-H. Figure 5 compares performance with different input documents for all NLP-enhanced tuning baselines. While the performance of Prior-Simple does not change with the input text, the performance of Prior-Main degrades as we switch to the smaller document. Prior-Main benefits from large document collections as redundant hints can partially make up for imprecise extractions. For the smaller input document, it does not extract any hints. DB-BERT, however, benefits from more specialized tuning hints. Using benchmark-specific input text, it converges to near-optimal solutions faster and ultimately finds a slightly better solution (using a higher value for the `shared_buffers` parameter, compared to Table 5, as proposed in the blog entry).

Finally, we scale up the data size. Figure 12 reports results for TPC-H with scaling factor 10 (and using the TPC-H specific tuning text).[15] Compared to Fig. 11, showing results for scaling factor one, it takes longer for DB-BERT to find near-optimal solutions. This is expected, as longer run times per benchmark evaluation reduce the number of DB-BERT's iterations per time unit. Compared to other baselines, DB-BERT finds significantly better solutions again.

## 10 Conclusion and outlook

We presented DB-BERT, a database tuning system that extracts tuning hints from text documents. Our experiments demonstrate that such hints lead to significantly better tuning results.

In future work, we will consider more diverse tuning objectives. Currently, DB-BERT is limited to optimizing metrics such as latency or throughput that can be easily measured. However, there are other, important metrics that are difficult to measure. For instance, many parameters (e.g., the `fsync` parameter in PostgreSQL) allow increasing performance if willing to accept a small risk of data loss. Database manuals typically contain warnings detailing such risks. We plan to extend DB-BERT to extract information on metrics that are difficult to measure from the manual. Thereby, it can support users in finding parameter settings that maximize performance while complying with constraints on other metrics.

The current DB-BERT version only supports extraction for a common but limited class of tuning hints. More precisely, it support absolute value recommendations and relative recommendations that depend on one single system property. In some cases, recommendations link multiple parameters together. For instance, a recommendation such as "We can use the formula below to calculate the optimal `work_mem` value for the database server: Total RAM $*$ 0.25 / `max_connections`"[16] is not supported. Expanding the scope to such hints requires extensions of the text extraction mechanism as well as of the reinforcement learning approach (since decisions for different parameters are not independent anymore). However, mining such hints seems particularly powerful as discovering dependencies between multiple parameters via trial runs is expensive.

Sometimes, tuning hints come with valuable context, restricting their scope to specific scenarios. For instance, the hint "Note that on Windows, large values for `shared_buffers` aren't as effective, and you may find better results keeping it relatively low and using the OS cache more instead"[17] refers to platforms with a Windows operating system. Beyond the operating system, hints may refer to specific hardware platforms (e.g., low parallelism versus high parallelism) or to specific workload types (e.g., analytical versus transactional workloads). For the moment, DB-BERT only exploits such context indirectly, by allowing users to retrieve documents that contain certain keywords (e.g., "analytical workload"). Exploiting context directly would require DB-BERT to recognize context and link it to properties of the current tuning scenario.

In summary, while the current DB-BERT version already benefits significantly from parsed tuning hints, we see various avenues for future research and improvements.

**Acknowledgements** This project is supported by NSF CAREER grant IIS-2239326 ("Mining Hints from Text Documents to Guide Automated Database Performance Tuning").



[15] Note that execution time for the best configuration increases slightly at the beginning for DDPG10. This cannot happen as long as we consider a single run. However, we average over a smaller set of runs that finished their first evaluation fast for the first data point, while the second data point averages over all runs.

[16] https://www.enterprisedb.com/postgres-tutorials/how-tune-postgresql-memory.

[17] https://amonrait.freshdesk.com/support/solutions/articles/6000252619-how-to-increase-max-connections.

## References

1. Aken, D.V., Pavlo, A., Gordon, G.J.: Automatic database management system tuning through large-scale machine learning. In SIGMOD. pp. 1009–1024. (2017)
2. Arora, S., Yang, B., Eyuboglu, S., Narayan, A., Hojel, A., Trummer, I., Re, C.: language models enable simple systems for generating structured views of heterogeneous data lakes. In: PVLDB. vol. 17(2), pp. 92–105. (2023)
3. Basu, D., Lin, Q., Chen, W., Vo, H.T., Yuan, Z., Senellart, P., Bressan, S.: Cost-model oblivious database tuning with reinforcement learning. In: LNCS. vol. 9261, pp. 253–268. (2015)
4. Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., Amodei, D.: Language models are few-shot learners. Adv. Neural Inform. Process. Syst. 1877–1901 (2020)
5. Chen, Z., Fan, J., Madden, S., Tang, N.: Symphony: towards natural language query answering over multi-modal data lakes. In CIDR. pp. 1–7. (2023)
6. Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H.W., Sutton, C., Gehrmann, S., Schuh, P., Shi, K. Tsvyashchenko, S., Maynez, J., Rao, A., Barnes, P., Tay, Y., Shazeer, N. Prabhakaran, V., Reif, E., Du, N., Hutchinson, B., Pope, R., Bradbury, J., Austin, J., Isard, M., Gur-Ari, G., Yin, P., Duke, T., Levskaya, A., Ghemawat, S., Dev, S., Michalewski, H., Garcia, X., Misra, V., Robinson, K., Fedus, L., Zhou, D., Ippolito, D., Luan, D., Lim, H., Zoph, B., Spiridonov, A., Sepassi, R., Dohan, D., Agrawal, S., Omernick, M., Dai, A.M., Pillai, T.S., Pellat, M., Lewkowycz, A., Moreira, E., Child, R., Polozov, O., Lee, K., Zhou, Z., Wang, X., Saeta, B., Diaz, M., Firat, O., Catasta, M., Wei, J., Meier-Hellstern, K., Eck, D., Dean, J., Petrov, S., Fiedel, N.: PaLM: Scaling Language Modeling with Pathways. In CoRR. pp. 1–87. (2022) arxiv:2204.02311
7. Devlin, J., Chang, M.W., Lee, K., Toutanova, K.: BERT: pre-training of deep bidirectional transformers for language understanding. In NAACL. pp. 4171–4186. (2019)
8. Ding, B., Das, S., Marcus, R., Wu, W., Chaudhuri, S., Narasayya, V.R.: AI meets AI: Leveraging query executions to improve index recommendations. In SIGMOD. pp. 1241–1258. (2019)
9. Doshi, L., Zhuang, V., Jain, G., Marcus, R., Huang, H., Altinbüken, D., Brevdo, E., Fraser, C.: Kepler: Robust Learning for Parametric Query Optimization. In: Proceedings of the ACM on Management of Data. vol. 1(1), pp. 1–25. (2023)
10. Floridi, L., Chiriatti, M.: GPT-3: its nature, scope, limits, and consequences. Minds Mach. **30**(4), 681–694 (2020)
11. Gatt, A., Krahmer, E.: Survey of the state of the art in natural language generation: core tasks, applications and evaluation. pp. 1–111. (2017) arXiv preprint arXiv:1703.09902
12. Giannakouris, V., Trummer, I.: Building Learned Federated Query Optimizers. In VLDB PhD Workshop (2022)
13. Guo, J., Zhan, Z., Gao, Y., Xiao, Y., Lou, J.-G., Liu, T., Zhang, D.: Towards complex text-to-SQL in cross-domain database with intermediate representation. In ACL. pp. 4524–4535. (2019)
14. Herzig, J., Nowak, P.K., Müller, T., Piccinno, F., Eisenschlos, J.M.: TAPAS: weakly supervised table parsing via pre-training. (2019)
15. Hilprecht, B., Binnig, C., Röhm, U.: Learning a partitioning advisor for cloud databases. In SIGMOD. pp. 143–157. (2020)
16. Howard, J., Ruder, S.: Universal Language Model Fine-tuning for Text Classification. In ACL. pp. 328–339. (2018)
17. Kanellis, K., Alagappan, R., Venkataraman, S.: Too many knobs to tune? Towards faster database tuning by pre-selecting important knobs. In HotStorage 2020 - 12th USENIX Workshop on Hot Topics in Storage and File Systems, co-located with USENIX ATC 2020, pp. 1–8. (2020)
18. Karagiannis, G., Saeed, M., Papotti, P., Trummer, I.: Scrutinizer: a mixed-initiative approach to large-scale. Data-driven claim verification. In: PVLDB. vol. 13(12), pp. 2508–2521. (2020)
19. Kayali, M., Lykov, A., Fountalis, I., Vasiloglou, N., Olteanu, D., Suciu, D.: CHORUS: foundation models for unified data discovery and exploration. In: CoRR. (2023) arxiv:2306.09610
20. Kraska, T., Li, T., Madden, S., Markakis, M., Ngom, A., Wu, Z., Yu, G.X.: Check out the big brain on BRAD: simplifying cloud data processing with learned automated data meshes. In: PVLDB. vol. 16(11), pp. 3293–3301. (2023)
21. Krishnan, S., Yang, Z., Goldberg, K., Hellerstein, J., Stoica, I.: Learning to optimize join queries with deep reinforcement learning. In aiDM. pp. 1–6. (2020)
22. Leis, V., Gubichev, A., Boncz, P., Kemper, A., Neumann, T.: How good are query optimizers, really?. In: PVLDB. vol. 9(3), pp. 204–215. (2015)
23. Li, G., Zhou, X., Cao, L.: AI meets database: AI4DB and DB4AI. In: Proceedings of the ACM SIGMOD International Conference on Management of Data. pp. 2859–2866. (2021)
24. Li, G., Zhou, X., Li, S., Gao, B.: QTune: a QueryAware database tuning system with deep reinforcement learning. In: PVLDB. vol. 12(12), pp. 2118–2130. (2018)
25. Lieber, O., Sharir, O., Lenz, B., Shoham, Y.: Jurassic-1: technical details and evaluation. Technical report (2021)
26. Lin, X.V., Socher, R., Xiong, C.: Bridging textual and tabular data for cross-domain Text-to-SQL semantic parsing. In EMNLP. pp. 4870–4888. (2020)
27. Luketina, J., Nardelli, N., Farquhar, G., Foerster, J., Andreas, J., Grefenstette, E., Whiteson, S., Rocktäschel, T.: A survey of reinforcement learning informed by natural language. In IJCAI. pp. 6309–6317. (2019)
28. Marcus, R., Negi, P., Mao, H., Zhang, C., Alizadeh, M., Kraska, T., Papaemmanouil, O., Tatbul, N.: Neo: a Learned query optimizer. In: PVLDB. vol. 12(11), pp. 1705–1718. (2018)
29. Marcus, T.R., Negi, P., Mao, H., Tatbul, N., Alizadeh, M., Kraska, B.: Making Learned Query Optimization Practical: In ACM SIGMOD Record. vol. 51, pp. 6–13. (2022)
30. Nemhauser, G., Wolsey, L.: Best algorithms for approximating the maximum of a submodular set function. Math. Oper. Res. **3**(3), 177–188 (1978)
31. Oracle. MySQL 8. 0 Reference Manual. 2021
32. Ortiz, J., Balazinska, M., Gehrke, J., Keerthi, S.S.: Learning state representations for query optimization with deep reinforcement learning. In DEEM. (2018)
33. Patibandla, P.: https://amplitude.engineering/how-a-single-postgresql-config-change-improved-slow-query-performance-by-50x-85593b8991b0 (2017)
34. Pavlo, A., Angulo, G., Arulraj, J., Lin, H., Lin, J., Ma, L., Menon, P., Mowry, T. C., Perron, M., Quah, I., Santurkar, S., Tomasic, A.,

Toor, S., Aken, D. V., Wang, Z., Wu, Y., Xian, R., Zhang, T.: Self-driving database management systems. In CIDR. pp. 1–6. (2017)
35. PERCONA. Tuning PostgreSQL parameters to optimize performance. (2018) https://www.percona.com/blog/2018/08/31/tuning-postgresql-database-parameters-to-optimize-performance/
36. Rajpurkar, P., Zhang, J., Lopyrev, K., Liang, P.: SQuad: 100,000+ questions for machine comprehension of text. In EMNLP. pp. 2383–2392. (2016)
37. Saeed, M., De Cao, N., Papotti, P.: Querying large language models with SQL. In: CoRR. (2023) arxiv:2304.00472
38. Saha, D., Floratou, A., Sankaranarayanan, K., Minhas, U.F., Mittal, A.R., Ozcan, F.: ATHENA: An ontology-driven system for natural language querying over relational data stores. In: VLDB. vol. 9(12), pp. 1209–1220 (2016)
39. Suri, S., Ilyas, I., Re, C., Rekatsinas, T.: Ember: no-code context enrichment via similarity-based keyless joins. In: PVLDB. vol. 15(3), pp. 699–712. (2021)
40. Sutton, R.S., Barto, A.G.: Reinforcement learning, second edition: An introduction. (2018)
41. Tang, N., Fan, J., Li, F., Tu, J., Du, X., Li, G., Madden, S., Ouzzani, M.: Rpt: Relational pre-trained transformer is almost all you need towards democratizing data preparation. In: PVLDB. vol. 14(8), pp. 1254–1261 (2021)
42. Thorne, J., Yazdani, M., Saeidi, M., Silvestri, F., Riedel, S., Halevy, A.: From natural language processing to neural databases. In: Proceedings of the VLDB Endowment. vol.14(6), pp. 1033–1039. (2021)
43. Trummer, I.: The case for NLP-enhanced database tuning: towards tuning tools that "read the manual". In PVLDB. (2021)
44. Trummer, I.: CodexDB: synthesizing code for query processing from natural language instructions using GPT-3 codex. In: PVLDB. vol. 15(11), pp. 2921–2928. (2022)
45. Trummer, I.: DB-BERT: a Database Tuning Tool that "Reads the Manual". In: SIGMOD. pp. 190–203. (2022)
46. Trummer, I.: From BERT to GPT-3 codex: harnessing the potential of very large language models for data management. In: PVLDB. vol. 15(12), pp. 3770–3773. (2022)
47. Trummer, I.: Can large language models predict data correlations from column names?. In: PVLDB. vol. 16(13), pp. 4310–4323. (2023)
48. Trummer, I., Wang, J., Maram, D., Moseley, S., Jo, S., Antonakakis, J.: SkinnerDB: regret-bounded query evaluation via reinforcement learning. In: SIGMOD. pp. 1039–1050. (2019)
49. Van Aken, D., Yang, D., Brillard, S., Fiorino, A., Zhang, B., Bilien, C., Pavlo, A.: An inquiry into machine learning-based automatic configuration tuning services on real-world database management systems. In: PVLDB. vol. 14(7), pp. 1241–1253. (2021)
50. Van Hasselt, H., Guez, A., Silver, D.: Deep reinforcement learning with double Q-Learning. In AAAI. pp. 2094–2100. (2016)
51. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł., Polosukhin, I.: Attention is all you need. Adv. Neural Inf. Process. Syst. 5999–6009 (2017)
52. Wang, J., Trummer, I., Basu, D.: UDO: universal database optimization using reinforcement learning. In: PVLDB. vol. 14(13), pp. 3402–3414. (2021)
53. Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Funtowicz, M., Davison, J., Shleifer, S., Platen, P.V., Ma, C., Jernite, Y., Plu, J., Xu, C., Scao, T.L., Gugger, S., Drame, M., Lhoest, Q., Rush, A.M.: Transformers: state-of-the-art natural language processing. pp. 38–45. (2020)
54. Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., Davison, J., Shleifer, S., von Platen, P., Ma, C., Jernite, Y., Plu, J., Xu, C., Le Scao, T., Gugger, S., Drame, M., Lhoest, Q., Rush, A.: Transformers: state-of-the-art natural language processing. In EMNLP. pp. 38–45. (2020)
55. Yang, Z., Chandramouli, B., Wang, C., Gehrke, J., Li, Y., Minhas, U.F., Larson, P.Å., Kossmann, D., Acharya, R.: Qd-tree: learning data layouts for big data analytics. In SIGMOD. pp. 193–208. (2020)
56. Yu, T., Zhang, R., Yang, K., Yasunaga, M., Wang, D., Li, Z., Ma, J., Li, I., Yao, Q., Roman, S., Zhang, Z., Radev, D.R.: Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, EMNLP 2018. pp. 3911–3921. (2020)
57. Zhang, J., Liu, Y., Zhou, K., Li, G., Xiao, Z., Cheng, B., Xing, J., Wang, Y., Cheng, T., Liu, L., Ran, M., Li, Z.: An end-to-end automatic cloud database tuning system using deep reinforcement learning. In SIGMOD. pp. 415–432. (2019)
58. Zhang, S., Roller, S., Goyal, N., Artetxe, M., Chen, M., Chen, S., Dewan, C., Diab, M.T., Li, X., Lin, X.V., Mihaylov, T., Ott, M., Shleifer, S., Shuster, K., Simig, D., Koura, P.S., Sridhar, A., Wang, T., Zettlemoyer, L.: OPT: open pre-trained transformer language models. In: CoRR. (2022) arxiv:2205.01068
59. Zhong, V., Xiong, C., Socher, R.: Seq2SQL: generating structured queries from natural language using reinforcement learning. In: CoRR. pp. 1–12. (2017) arxiv:1709.00103