Appendix

OUTLINE

This supplemental material is organized as follows:
A.1 More details about OnlineTune Architecture.
A.2 High-Level Algorithm.
A.3 Details about subspace adaptation.
A.4 More details and results about experiments.

A1 MORE DETAILS ABOUT ONLINETUNE ARCHITECTURE

System Architecture. Figure A1 presents the architecture of OnlineTune. The left part shows the online database system in the cloud environment that runs ever-changing workloads. The right part represents the OnlineTune server deployed in the backend tuning cluster. OnlineTune maintains a data repository that stores the historical observations $\{< c_i, \theta_i, y_i >\}_1^t$ from the previous tuning iterations, which could be initially empty. The controller monitors the states of tuning tasks and transfers the data between the database side and OnlineTune server side. The main parts of OnlineTune run on the server side, whose resource consumption doesn't affect the online database. The context featurization module is deployed in the database instance for data privacy concerns.

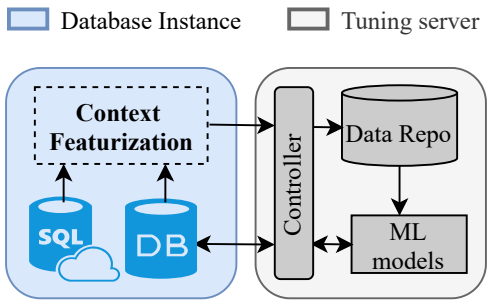

Figure A1: OnlineTune Architecture.

Discussion. OnlineTune is a reactive approach: it featurizes the online workloads at the beginning of an iteration and applies a configuration for the later time of the iteration. Since changes in real-world workloads are gradual, the little lagging can be acceptable. A predictive module for the context might help. However, it still assumes gradual and predictable changes based on historical data, and an inaccurate prediction module could greatly affect the tuner's performance.

A2 TOP-LEVEL ALGORITHM

Algorithm 3 presents the top-level algorithm. OnlineTune first queries the default configuration and its performance to form the initial safety set and initializes the data repository (Line1–3). The safety threshold is set as the default performance. At the beginning of an iteration, OnlineTune collects the incoming workload queries and corresponding optimizer's estimations to calculate context vector c_i (Line 5, Section 5.1). OnlineTune loads a prediction model m_{i-1}^n selected by a SVM model that inputs the context vector and outputs a cluster label n (Line 6, Section 5.3). Then, a configuration subspace is initialized or adapted (Line 7, Section 6.1). OnlineTune discretizes the configuration subspace and assesses the safety of unevaluated configurations based on the black-and-white prior knowledge to form a safe candidate set (safety set) (Line 8, Section 6.2). Next, OnlineTune recommends a configuration θ_i within the safety set to trade-off exploitation (i.e., making decisions based on existing knowledge) and exploration (i.e., acquiring new knowledge or expanding the safety set) (Line 9, Section 6.3). The configuration θ_i is applied to the online database, and the database performance y_i during the tuning interval is collected. Finally, OnlineTune updates the prediction model and data repository with $\{\theta_i, c_i, y_i\}$ (Lines 12 to 11). OnlineTune determines whether to re-cluster or not ((Line 13, see Algorithm 1 for details). If needing re-cluster, OnlineTune clusters the observations based on $\{c_i\}_1^t$, fits prediction models for each cluster, and re-train the SVM model, as illustrated in Section 5.3.

Algorithm 3: Top-Level Algorithm of OnlineTune

Output: Configuration recommendation adaptively with changing environment

1 Featurize the environment factor, get context c_0.
2 Query default configuration θ_0 and get its performance y_0.
3 Initialize a data repository H_0 with $< c_0, \theta_0, y_0 >$.
4 **for** $i \leftarrow 1$ *to* K **do**
5 Featurize and get context c_i.
6 Select a model m_{i-1}^n, where $n = SVM(c_i)$.
7 Θ_i^n = `Subspace_Adaptation` $(\Theta_{i-1}^n, H_{i-1})$.
8 Generate safe candidates $S_i^n \in \Theta_i^n$.
9 Select a configuration θ_i within S_i^n .
10 Apply θ_i and evaluate its performance y_i.
11 H_i = `Append`$(H_{i-1}, < \theta_i, c_i, y_i >)$.
12 Fit prediction model m_i^n on H_i.
13 `Offline_Clustering`(H_i).

A3 DETAILS ABOUT SUBSPACE ADAPTATION

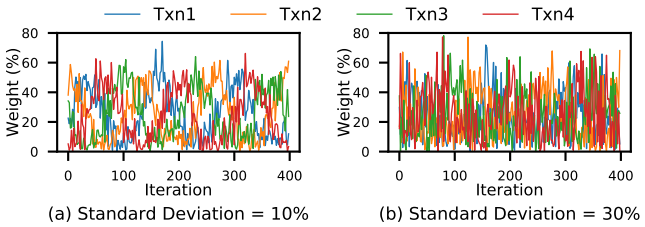

Figure A2: Transaction weight for dynamic workloads.

Table A1: Average time breakdown for one tuning iteration on JOB workload.

	Featurization	Model Selection	Model Update	Subspace Adaptation	Safety Assessment	Candidate Selection	Apply & Evaluation
Average Time	0.0577s	0.0259s	1.3583s	0.2695s	0.1354s	0.0278s	178.1254s
Percentage	0.03%	0.01%	0.75%	0.15%	0.08%	0.02%	98.96%

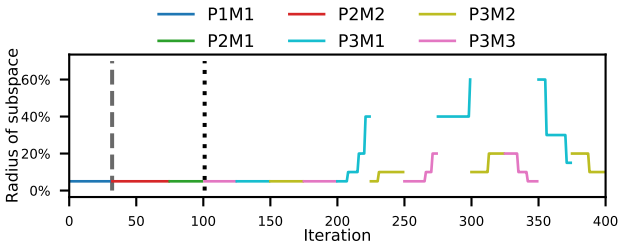

Figure A3: The radius of subspaces over iteration.

A3.1 More Rationale about Subspace Restriction

Given a configuration space Θ and a contextual GP model, we can discretize Θ to obtain a set of candidates and assess the safety of each candidate based on the estimation of the GP model. However, the direct operation over the whole space suffers from the curse of dimensionality. First, the candidates need to be close to each other in order to use the GP to generalize safety. When tuning more than a couple of knobs, it is computationally affordable to conduct a fine discretization over the whole configuration space [37]. In addition, the database performance function is often complex, making the estimation of the GP model problematic for high dimensional configuration (more than ten parameters) [55]. To deal with those challenges, we draw inspiration from a class of trust-region methods in stochastic optimization [75]. These methods use a linear and quadratic prediction model inside a trust region, which is often a sphere or a polytope centered at the best observation. Intuitively, while linear and quadratic models are likely to be inadequate to model globally, they can be accurate in a sufficiently small trust region [20]. However, the linear models struggle to handle noisy observations and require small trust regions to provide accurate modeling behavior. Naturally, we use contextual GP as the prediction model to describe the function $f(\theta, c)$ and restrict OnlineTune's optimization space in the trust region (i.e., configuration subspace). The optimization can be solved efficiently within the trust region, and the GP model's estimation can be trusted in the trust region [20].

A3.2 Direction Oracles for Line Region

OnlineTune implements two strategies to generate the directions, including random direction and important direction.

Random direction. A strategy is to pick the direction uniformly (random direction), increasing the exploration.

Important direction. OnlineTune also chooses the directions aligned with the important configuration knob (important direction), inspired by the important knobs pre-selecting procedure before tuning used by existing configuration tuning approaches [10, 22, 35]. It is empirically shown that restricting the optimization space in several important knobs can largely reduce the tuning iterations and achieve similar improvement [35]. The importance of knobs is quantified by Fanova [31], a line-time approach for assessing feature importance. However, detecting important knobs needs thousands of evaluation samples for a given workload. And fixing the configuration space wrongly (e.g., filtering important knobs) will severely hinder the optimization. OnlineTune's adaptation of subspace solves this problem by adjusting the subspace over iterations.

A random direction is chosen if the performance improvement in the previous hypercube region is lower than a threshold (exploration). Otherwise, an important direction is chosen by sampling from the top-5 important knobs (exploitation). The importance of knobs is updated with the increasing observations.

A4 MORE DETAILS ABOUT EXPERIMENTS

A4.1 The Details for Setup

Setting for Workload Featurization. For workload featurization in OnlineTune, we use a one-layer bidirectional LSTM as encoder, and the weights of encoder and decoder are shared. The dimension of query embedding is set to 6. We use queries from 10 workloads in OLTP-Bench, JOB, and the real-world workloads to train the auto-encoder. The training data has ~ 40k queries and 403 distinct templates. Note that the encoder's training set can be extended to enhance generalizability. For example, Zolaktaf et al. use Sloan Digital Sky Survey (SDSS) query workload, which contains ~ 618k queries[79].

Knobs Selection. We pick the tuning knobs by selecting the union of the important features based on the historical observation data. DBA removes the knobs whose change could cause severe consequences and inconsistent behaviors (e.g., innodb_flush_log_at_trx_commit, sync_binlog). With such a black tuning list, the knobs selection can be conducted automatically. As cloud providers, we only select the important configuration knobs to tune once and use them as a tuning set for all cloud workloads.

Performance Evaluation. When evaluating the performance of the recommended configurations, OnlineTune first queries the knobs' values and skip the performance statistics until the knobs' values are updated. After the values are updated, we also skip the performance statistics in the first 30 seconds for a stable measurement as most benchmark tools do the same things (warm-up).

Transaction Weights for OLTP Workloads. We construct dynamic OLTP workloads by varying the transaction weight. The weights are sampled from a normal distribution with a sine function of iterations as mean and a 10% standard deviation, as shown in figure A2 (a). We also conduct experiments with more drastic changes of workload (Figure A2 (b)) and more hardware instance types to evaluate the robustness of OnlineTune in Section A4.4 and Section A4.5.

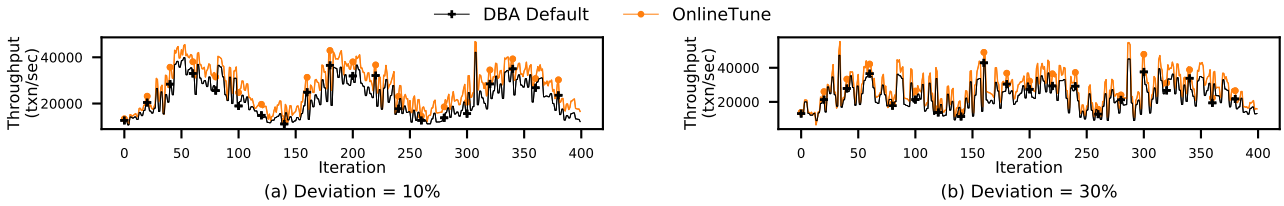

Figure A4: Tuning Twitter workload with difference various of changes.

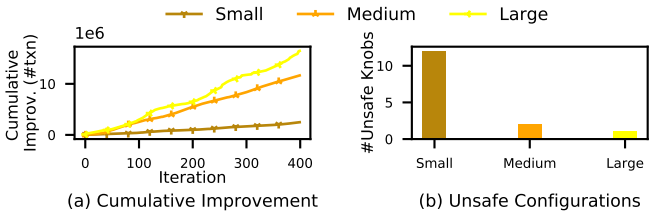

Figure A5: Tuning TPC-C workload on different instances.

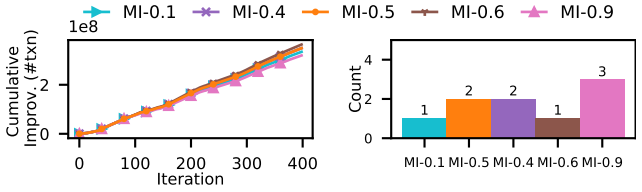

Figure A6: Tuning Twitter with different MI thresholds.

A4.2 Resource Consumption of Backend Tuning Cluster

In the paper, we analyze the resource consumption of context featurization deployed in the database instance. Now we analyze the resource consumption of the backend tuning cluster where OnlineTune's other components (model selection, model update, and safe configuration recommendation) are deployed. We evaluate on a tuning server with 4 vCPU and 16GB RAM. We observe that the average CPU usage is about 0.39% when running one tuning task for half one hour since the tuner's computation time is within 3.79 seconds in the tuning interval of 180 seconds. Table A1 presents the detailed time breakdown on average. In practice, the tuning server can be scaled up automatically when running multiple tuning tasks.

A4.3 More Details About Case Study

Figure A3 presents the radius of subspace maintained by each model (indicated by different colors). The radius of a subspace (hypercube region) is first set as a base value (5% ranges of each dimension) and adjusted according to the tuning performance. OnlineTune alternates between hypercube regions and line regions. When the corresponding subspace is a line region, its radius is plotted unchanged. OnlineTune doubles the subspace radius when consecutively recommending configurations better than the previous one or otherwise halves the radius, We observe that after 200 iterations, the subspace radius initially increases (favoring explorations) and then decreases (favoring exploitation).

A4.4 Varying Variance of Changes

OnlineTune recommends configurations based on the context feature extracted from the dynamic environment. In practice, the change of workload and the underlying data is gradual, and OnlineTune generalizes over similar contexts. When the variance of changes increases, safely optimizing the online database becomes more challenging. We evaluate the performance of OnlineTune on tuning Twitter workload sampled with different variance: 10% and 30% deviation as shown in Figure A2, and Figure A4 presents OnlineTune's tuning performance. The line of the DBA default's performance is more fluctuating in Figure A4 (b), as the workload changes more drastically. When tuning the workload with 10% deviation, OnlineTune achieves 21.8% improvement on cumulative performance compared to the DBA default with two unsafe recommendations, and 18.7% improvement with five unsafe recommendations when tuning the workload with 30% deviation, which shows the robustness of OnlineTune on workload changes.

A4.5 Varying Hardware Types

We apply OnlineTune to tune dynamic workload Twitter on three different hardware instances: (1) Small, an instance with 4 vCPU, 8GB RAM, (2) Medium: an instance with 8 vCPU, 16GB RAM, and (3) Large: an instance with 16 vCPU, 32GB RAM. Figure A5 shows the cumulative improvement and statistics of unsafe recommendations. System failures are omitted since no failure occurs during tuning. We observe that OnlineTune safely tunes the databases and achieves performance improvement. In the small instance, the number of unsafe configurations is slightly larger since the limited machine capacity makes it more challenging to ensure performance above the safety threshold. The tuning configuration spaces of the three instances are the same. And the safe area on the small instance is more constrained. When tuning the small instance, the tuner has more chances to explore the unsafe area. As the instance's capacity increases, the number of unsafe configurations decreases, and the cumulative improvement increases.

A4.6 Varying MI Thresholds

Given context feature trace, OnlineTune controls the frequency of re-clustering via the MI threshold. OnlineTune maintains the existing clustering and a simulated new clustering, and quantifies their difference through mutual information-based (MI) score. The re-clustering is triggered when the MI score between the two clusterings is smaller than the MI threshold. We conduct sensitivity analysis on the setting of MI threshold, as shown in Figure A6. We observe that OnlineTune is not very sensitive to the values

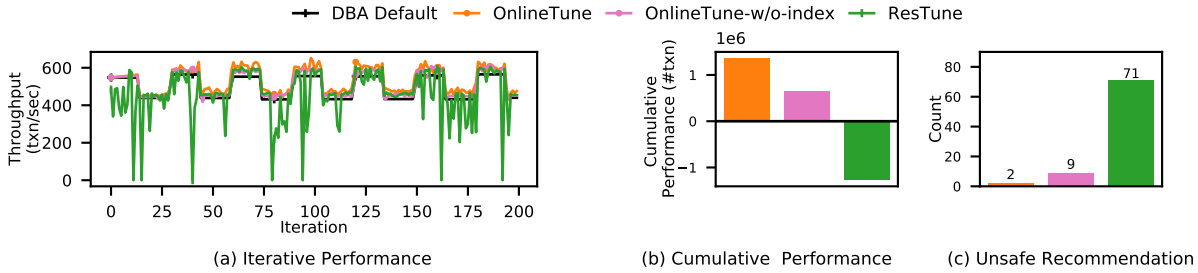

Figure A7: Running TPC-C with index change.

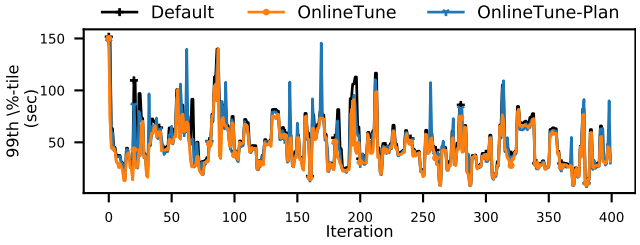

Figure A8: Iterative performance on JOB workload: The lower is the better.

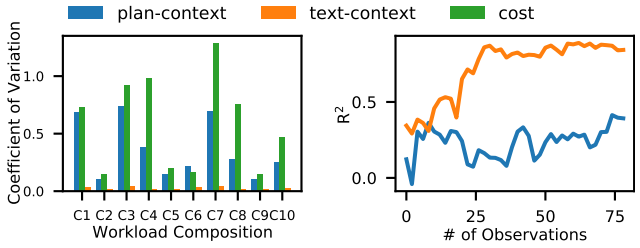

Figure A9: Coefficient of variance and model accuracy.

of MI threshold. This is because the number of clusters is mainly affected by the content of context features clustered by the DBSCAN algorithm. When setting the MI threshold to 0.4 or 0.5, the re-clustering occurs three times, at iteration 86 (2 clusters), iteration 202 (4 clusters), and iteration 292 (9 clusters). A larger MI threshold leads to more frequent re-clustering. And a MI threshold equalling one indicates that OnlineTune conducts the re-clustering every time the simulated new clustering and the existing one are different. When setting the MI threshold to 0.9 or 1, the re-clustering occurs eight times and also ends up with 9 clusters.

A4.7 Adaptability on Index Change

OnlineTune encodes whether an index is used in its context feature. We conduct an experiment with explicit index dropping/building operations on TPC-C workload. We drop a secondary index on the CUSTOMER table for the (C_W_ID,C_D_ID,C_LAST,C_FIRST) columns, which inhibits the performance of TPC-C during iteration 15. Then, we build the index during iteration 30. The dropping/building operations are repeated every 15 iterations. We compare the three baselines: (1) OnlineTune, (2) OnlineTune-w/o-index: removing the encoding for index, (3) ResTune, an offline method with relatively good performance. Figure A7 presents the results. We observe that ResTune causes seven system failures while none occur when applying OnlineTune and OnlineTune-w/o-index. And ResTune's tuning causes the negative performance improvement, inferior to OnlineTune and OnlineTune-w/o-index, as shown in Figure (b). OnlineTune achieves better performance than OnlineTune-w/o-index and has fewer unsafe recommendations, illustrating the effect of encoding the dynamicity.

A4.8 Analysis of Workload Featurization

We compare the workload featurization using *text-context* (context using query text and its cardinality, as in OnlineTune) and *plan-context* (context using plan, including the operators and the costs). For *plan-context*, we use the same encoding method as QTune. The encoding is a $4 + |T| + |O|$ dimensional vector, where $|T|$ is the number of tables in the database, and $|O|$ is the number of operations. The first four features capture the query types. For an insert/select/update/delete query, the corresponding value is 1, 0 otherwise. The latter $|T|$ features denote tables. If the query contains the table, the corresponding value is 1, 0 otherwise. The last $|O|$ features are the cost estimation for each operator. Note that an operator may appear in different nodes of the plan, and the costs of the same operator are summed up as the corresponding cost value.

We first conduct an end-to-end tuning experiments on dynamic JOB with the same setting as in Section 7.1.1, comparing using *text-context* (OnlineTune) and *plan-context* (OnlineTune-plan). Figure A8 presents the results. We observe that OnlineTune using *text-context* has better performance. OnlineTune has 8.73% improvement than OnlineTune-plan on cumulative performance. And OnlineTune-Plan has 31 unsafe configurations while OnlineTune has 8 unsafe configurations. The reason is as follows. Query plan contains extra information than the text and cardinality, such as the operators and corresponding costs. The query text might remain unchanged, but the optimizer might decide to make plan changes based on configurations. The extra information in the plan is affected by OnlineTune's configuration, and precisely, configuration applied at the previous iteration. This is because OnlineTune featurizes the context and applies a promising configuration θ_t at the beginning of an iteration t. Then OnlineTune evaluates the performance of θ_t during the iteration and obtains y_t. For a given workload composition, its query texts are unchanged, but the operators and costs in its query plans may vary with the configurations

θ_{t-1}. If using the extra information in plans as context (hereinafter called *plan-context*), c_t is affected by θ_{t-1}. And the influence of θ_{t-1} can be viewed as noise for the mapping from $\langle c_t, \theta_t \rangle$ to y_t, since θ_{t-1} is determined by OnlineTune's previous configuring.

To further substantiate our argument, we use an experiment to prove the variance of *plan-context* and compare the model accuracy using the two types of contexts. We use JOB workload to construct ten different workload compositions. For each composition, we randomly sample 20 configurations. Then we follow the workflow of OnlineTune iteratively: (1) conduct context featurization and obtain *text-context* and the *plan-context*; (2) apply one sampled configuration and evaluate its performance. We observe that for each composition, the *text-context* is nearly unchanged since the query text is fixed, and JOB is a read-only workload not causing any underlying data changes (no cardinality estimation change). However, the *plan-context* varies dramatically due to the change of estimated costs influenced by the previous configurations. We use coefficient of variation (CV) to measure the variation of *text-context* and *plan-context* for each workload composition, as shown in Figure A9 (left). CV is the ratio of the standard deviation to the mean. The higher the coefficient of variation, the greater the level of dispersion around the mean. We calculate CV for each feature and present the averaged value. For reference, we also plot the CV of the total cost. We observe that *plan-context* has a large CV, correlated with cost, while *text-context* has CV near to zero.

We then compare the modelling accuracy using *text-context* and *plan-context*. OnlineTune utilizes contextual GP to learn the mapping from $\langle c_t, \theta_t \rangle$ to y_t. In this experiment, we use the previously collected observations to fit the two kinds of contextual GP models (one based on *text-context* and one based on *plan-context*) and compare their model accuracy. We use 5% observations as validation data and 95% observations as test data. Figure A9 (right) presents the iterative coefficient of determination (R^2). The higher values indicate the better. We observe that model fit on the observations with *text-context* has higher accuracy. This is because *plan-context* is affected by previous random configurations θ_{t-1}, which is noisy for modeling the mapping from $\langle c_t, \theta_t \rangle$ to y_t.