Cost-Model Oblivious Database Tuning with Reinforcement Learning

Debabrota Basu[1(✉)], Qian Lin[1], Weidong Chen[1], Hoang Tam Vo[3], Zihong Yuan[1], Pierre Senellart[1,2], and Stéphane Bressan[1]

[1] School of Computing, National University of Singapore, Singapore, Singapore
debabrota.basu@u.nus.edu
[2] Institut Mines–Télécom, Télécom ParisTech, CNRS LTCI, Paris, France
[3] SAP Research and Innovation, Singapore, Singapore

Abstract. In this paper, we propose a learning approach to adaptive performance tuning of database applications. The objective is to validate the opportunity to devise a tuning strategy that does not need prior knowledge of a cost model. Instead, the cost model is learned through reinforcement learning. We instantiate our approach to the use case of index tuning. We model the execution of queries and updates as a Markov decision process whose states are database configurations, actions are configuration changes, and rewards are functions of the cost of configuration change and query and update evaluation. During the reinforcement learning process, we face two important challenges: not only the unavailability of a cost model, but also the size of the state space. To address the latter, we devise strategies to prune the state space, both in the general case and for the use case of index tuning. We empirically and comparatively evaluate our approach on a standard OLTP dataset. We show that our approach is competitive with state-of-the-art adaptive index tuning, which is dependent on a cost model.

1 Introduction

In a recent SIGMOD blog entry [10], Guy Lohman asked "Is query optimization a 'solved' problem?". He argued that current query optimizers and their cost models can be critically wrong. Instead of relying on wrong cost models, the author and his colleagues have proposed in [19] a *learning* optimizer.

In this paper, we propose a learning approach to performance tuning of database applications. By performance tuning, we mean selection of an optimal physical database configuration in view of the workload. In general, configurations differ in the indexes, materialized views, partitions, replicas, and other parameters. While most existing tuning systems and literature [6,17,18] rely on a predefined cost model, the objective of this work is to validate the opportunity for a tuning strategy to do without.

To achieve this, we propose a formulation of database tuning as a *reinforcement learning* problem (see Sect. 3). The execution of queries and updates is modeled as a Markov decision process whose states are database configurations,

Q. Chen et al. (Eds.): DEXA 2015, Part I, LNCS 9261, pp. 253–268, 2015.
DOI: 10.1007/978-3-319-22849-5_18

actions are configuration changes and rewards are functions of the cost of configuration change and query/update evaluation. This formulation does not rely on a pre-existing cost model, rather it learns it.

We present a solution to the reinforcement learning formulation that tackles the curse of dimensionality (Sect. 4). To do this, we reduce the search space by exploiting the quasi-metric properties of the configuration change cost, and we approximate the cumulative cost with a linear model.

We instantiate our approach to the use case of index tuning (Sect. 5). We use this case to demonstrate the validity of a cost-model oblivious database tuning with reinforcement learning, through experimental evaluation on a TPC-C workload [14] (see Sect. 6). We compare the performance with the *Work Function Index Tuning* (WFIT) Algorithm [18]. Results show that our approach is competitive yet does not need to know a cost model.

Related work is discussed in Sect. 2.

2 Related Work

Our work is intertwined with mainly two lines of research. Our methodology is designed to deal with the problem of automated database configuration. Using our approach described in Sect. 4, we have proposed COREIL, an algorithm to solve the problem of index tuning. Traditionally, most of the works proposed in the field of automated database configuration are conducted in an offline manner. In offline methodologies, database administrators identify and update representative workloads from the database queries based on these representative workloads, new database configurations are realized to create new beneficial indexes [1], smart vertical partition for reducing I/O costs [16], or possibly for engendering a combination of index selection, partitioning and replication for both stand-alone databases [11] and parallel databases [2].

But with increasing complexity and agility of database applications and the introduction of modern database environments such as database as a service, the aforementioned tasks of database administrators are becoming more tedious and problematic. Therefore it is desirable to design automated solutions of the database design problem that are able to continuously monitor the incoming queries or the changes in workload and can react readily by adapting the database configuration. An online approach for physical design tuning is proposed in [6], that progressively chooses an optimal solution at each step through case-by-case analysis of potential benefits. Similarly, [17] proposes a self-regulating framework for continuous online physical tuning where effective indexes are created and deleted in response to the shifting workload. In one of the most recent proposals for semi-automated index tuning, WFIT [18], the authors have proposed a method based on the Work function algorithm and the feedbacks from manual changes of configurations. To evaluate the cost of executing a query workload with the new indexes as well as the cost of configuration transition, i.e. for profiling indexes' benefit, most of the aforementioned online algorithms like WFIT exploit the what-if optimizer [7] which returns such estimated costs.

As COREIL is able to learn the estimated cost of queries gradually through subsequent iterations, it is applicable to a wider range of database management systems which may not implement what-if-optimizer or expose its interface to the users.

For designing and tuning online automated databases, our proposed approach uses the more general structure of *reinforcement learning* [20] that offers a rich pool of techniques available in literature. Markov decision processes (MDPs) [13] are one such model where each action leads to a new state and a given reward according to a probability distribution that must be learned. On the basis of such a cumulative reward, these processes decide the next action to perform for the optimal performance. Though use of Markov decision process for modelling data cleaning tasks has been proposed in [4], its application in data management is limited because of typically huge state space and complex structures of data in each state (in our case, indexes). Some recent research works like [3] have also approached dynamic index selection based on data mining and optimization algorithms. But in our proposed method, COREIL, we tackle the issues of using reinforcement learning in database applications. Other complications like delayed rewards obtained after a long sequence of state transitions and partial observability [21] of current state due to uncertainty, are also less prevalent in the proposed structure of COREIL.

3 Problem Definition

Let R be a logical database schema. We can consider R to be the set of its possible database instances. Let S be the set of physical database configurations of instances of R. For a given database instance, two configurations s and s' may differ in the indexes, materialized views, partitions, replicas, and other parameters. For a given instance, two configurations will be logically equivalent if they yield the same results for all queries and updates.

The cost of changing configuration from $s \in S$ to $s' \in S$ is denoted by the function $\delta(s, s')$. The function $\delta(s, s')$ is not necessarily symmetric as the cost of changing configuration from s to s' and the reverse may not be the same. On the other hand, it is a non-negative function and also verifies the identity of indiscernibles (there is no free configuration change) and the triangle inequality (it is always cheaper to do a direct configuration change). Therefore, it is a quasi-metric on S.

Let Q be a workload set, defined as a schedule of queries and updates (for brevity, we refer to both as queries). Without significant loss of generality, we consider the schedule to be sequential and the issue of concurrency control orthogonal to the current presentation. Query q_t is the t^{th} query in the schedule, which is executed at time t.

The cost of executing query $q \in Q$ on configuration $s \in S$ is denoted by the function $cost(s, q)$. We model a query q_t as a random variable, whose generating distribution may not be known *a priori*: q_t is only observable at time t.

Let s_0 be the initial configuration of the database. At any time t the configuration is changed from s_{t-1} to s_t with the following events in order:

1. Arrival of query q_t. We call $\hat{q}_t$ the observation of q_t at time t.
2. Choice of the configuration $s_t \in S$ based on $\hat{q}_1, \hat{q}_2, \ldots, \hat{q}_t$ and s_{t-1}.
3. Change of configuration from s_{t-1} to s_t. If no configuration change occurs at time t, then $s_t = s_{t-1}$.
4. Execution of query $\hat{q}_t$ under the configuration s_t.

The cost of configuration change and query execution at time t, referred as the *per-stage cost*, is

$$C(s_{t-1}, s_t, \hat{q}_t) := \delta(s_{t-1}, s_t) + cost(s_t, \hat{q}_t)$$

We can phrase in other words the stochastic decision process of choosing the configuration changes as a *Markov decision process* (MDP) [13] where states are database configurations, actions are configuration changes, and penalties (negative rewards) are the per-stage cost of the action. Note that transitions from one state to another on an action are deterministic (in contrast to the general framework of MDPs, there is no uncertainty associated with the new configuration when a configuration change is decided). On the other hand, penalties are both *stochastic* (they depend on the query, a random variable) and *uncertain* (the cost of a query in a configuration is not known in advance, in the absence of a reliable cost model).

Ideally, the problem would be to find the sequence of configurations that minimizes the sum of future per-stage costs; of course, assuming an infinite horizon [20], this sum is infinite. One practical way to circumvent this problem is to introduce a *discount factor* γ that gives more importance to immediate costs than to costs distant in the future, and to try and minimize a *cumulative cost* defined with γ. Under Markov assumption, a sequence of configuration changes is determined by a *policy* $\pi\colon S \times Q \to S$, which, given the current configuration s_{t-1} and a query $\hat{q}_t$, returns a configuration $s_t := \pi(s_{t-1}, \hat{q}_t)$.

We define the *cost-to-go* function V^π for a policy π as:

$$V^\pi(s) := \mathbb{E}\left[\sum_{t=1}^{\infty} \gamma^{t-1} C(s_{t-1}, s_t, \hat{q}_t)\right] \text{ satisfying } \begin{cases} s_0 = s \\ s_t = \pi(s_{t-1}, \hat{q}_t), t \geq 1 \end{cases} \quad (1)$$

where $0 < \gamma < 1$ is the discount factor. The value of $V^\pi(s)$ represents the expected cumulative cost for the following policy π from the current configuration s.

Let $\mathcal{U}$ be the set of all policies for a given database schema. Our problem can now be formally phrased as to minimize the expected cumulative cost, i.e., to find an optimal policy π^* such that $\pi^* := \arg\min_{\pi \in \mathcal{U}} V^\pi(s_0)$.

4 Adaptive Performance Tuning

4.1 Algorithm Framework

In order to find the optimal policy π^*, we start from an arbitrary policy π, compute an estimation of its cost-to-go function, and incrementally attempt

to improve it using the current estimate of the cost-to-go function $\overline{V}$ for each $s \in S$. This strategy is known as *policy iteration* [20] in reinforcement learning literature.

Assuming the probability distribution of q_t is known in advance, we improve the cost-to-go function $\overline{V}^{\pi_t}$ of the policy π_t at iteration t using

$$\overline{V}^{\pi_t}(s) = \min_{s' \in S} \left(\delta(s, s') + \mathbb{E}\left[cost(s', q)\right] + \gamma \overline{V}^{\pi_{t-1}}(s') \right) \tag{2}$$

We obtain the updated policy as $\arg\min_{\pi_t \in \mathcal{U}} \overline{V}^{\pi_t}(s)$. The algorithm terminates when there is no change in the policy. The proof of optimality and convergence of policy iteration can be found in [12].

Unfortunately, policy iteration suffers from several problems. First, there may not be any proper model available beforehand for the cost function $cost(s, q)$. Second, the curse of dimensionality [12] makes the direct computation of $\overline{V}$ hard. Third, the probability distribution of queries is not assumed to be known *a priori*, making it impossible to compute the expected cost of query execution $\mathbb{E}\left[cost(s', q)\right]$.

Algorithm 1. Algorithm Framework

1: Initialization: an arbitrary policy π_0 and a cost model C_0
2: Repeat till convergence
3: $\quad \overline{V}^{\pi_{t-1}} \leftarrow$ approximate using a linear projection over $\phi(s)$
4: $\quad C^{t-1} \leftarrow$ approximate using a linear projection over $\eta(s, \hat{q}_t)$
5: $\quad \pi_t \leftarrow \arg\min_{s \in S'} \left(C^{t-1} + \gamma \overline{V}^{\pi_{t-1}}(s) \right)$
6: End

The basic framework of our algorithm is shown in Algorithm 1. Initial policy π_0 and cost model C_0 can be intialized arbitrarily or using some intelligent heuristics. In line 5 of Algorithm 1, we have tried to overcome the issues at the root of the curse of dimensionality by juxtaposing the original problem with approximated per-stage cost and cost-to-go function. Firstly, we map a configuration to a vector of associated feature $\phi(s)$. Then, we approximate the cost-to-go function by a linear model $\theta^T \phi(s)$ with parameter θ. It is extracted from a reduced subspace S' of configuration space S that makes the search for optimal policy computationally cheaper. Finally, we learn the per-stage cost $C(s, s', \hat{q})$ by a linear model $\zeta^T \eta(s, \hat{q})$ with parameter ζ. This method does not need any prior knowledge of the cost model, rather it learns the model iteratively. Thus, we have resolved shortcomings of policy iteration and the need of predefined cost model for the performance tuning problem in our algorithm. These methods are depicted and analyzed in the following sections.

4.2 Reducing the Search Space

To reduce the size of search space in line 5 of c 1, we filter the configurations that satisfy certain necessary conditions deduced from an optimal policy.

Proposition 1. *Let s be any configuration and $\hat{q}$ be any observed query. Let π^* be an optimal policy. If $\pi^*(s, \hat{q}) = s'$, then $cost(s, \hat{q}) - cost(s', \hat{q}) \geq 0$. Furthermore, if $\delta(s, s') > 0$, i.e., if the configurations certainly change after query, then $cost(s, \hat{q}) - cost(s', \hat{q}) > 0$.*

Proof. Since $\pi^*(s, \hat{q}) = s'$, we have

$$\begin{aligned} & \delta(s, s') + cost(s', \hat{q}) + \gamma V(s') \\ \leq\ & cost(s, \hat{q}) + \gamma V(s) \\ =\ & cost(s, \hat{q}) + \gamma \mathbb{E}\left[\min_{s''} \left(\delta(s, s'') + cost(s'', \hat{q}) + \gamma V(s'')\right)\right] \\ \leq\ & cost(s, \hat{q}) + \gamma \delta(s, s') + \gamma V(s'), \end{aligned}$$

where the second inequality is obtained by exploiting triangle inequality $\delta(s, s'') \leq \delta(s, s') + \delta(s', s'')$, as δ is a quasi-metric on S.

This infers that

$$cost(s, \hat{q}) - cost(s', \hat{q}) \geq (1 - \gamma)\delta(s, s') \geq 0.$$

The assertion follows. □

By Proposition 1, if π^* is an optimal policy and $s' = \pi^*(s, \hat{q}) \neq s$, then $cost(s, \hat{q}) > cost(s', \hat{q})$. Thus, we can define a reduced subspace as

$$S_{s,\hat{q}} = \{s' \in S \mid cost(s, \hat{q}) > cost(s', \hat{q})\}.$$

Hence, at each time t, we can solve

$$\pi_t = \underset{s \in S_{s_{t-1}, \hat{q}_t}}{\arg\min} \left(\delta(s_{t-1}, s) + cost(s, \hat{q}_t) + \gamma \overline{V}^{\pi_{t-1}}(s)\right). \tag{3}$$

Next, we design an algorithm that converges to an optimal policy through searching in the reduced set $S_{s,\hat{q}}$.

4.3 Modified Policy Iteration with Cost Model Learning

We calculate the optimal policy using the *least square policy iteration* (LSPI) [9]. If for any policy π, there exists a vector $\boldsymbol{\theta}$ such that we can approximate $V^{\pi}(s) = \boldsymbol{\theta}^T \boldsymbol{\phi}(s)$ for any configuration s, then LSPI converges to the optimal policy. This mathematical guarantee makes LSPI an useful tool to solve the MDP as defined in Sect. 3. But the LSPI algorithm needs a predefined cost model to update the policy and evaluate the cost-to-go function. It is always not obvious that any form of cost model would be available and as mentioned in Sect. 1, pre-defined cost models may be critically wrong. This motivates us to develop another form of the algorithm, where the cost model can be equivalently obtained through learning.

Algorithm 2. Recursive least squares estimation.

1: **procedure** RLSE($\hat{\epsilon}^t, \overline{B}^{t-1}, \boldsymbol{\zeta}^{t-1}, \boldsymbol{\eta}^t$)
2: $\gamma^t \leftarrow 1 + (\boldsymbol{\eta}^t)^T \overline{B}^{t-1} \boldsymbol{\eta}^t$
3: $\overline{B}^t \leftarrow \overline{B}^{t-1} - \frac{1}{\gamma^t}(\overline{B}^{t-1}\boldsymbol{\eta}^t(\boldsymbol{\eta}^t)^T\overline{B}^{t-1})$
4: $\boldsymbol{\zeta}^t \leftarrow \boldsymbol{\zeta}^{t-1} - \frac{1}{\gamma^t}\overline{B}^{t-1}\boldsymbol{\eta}^t\hat{\epsilon}^t$
5: **return** $B^t, \boldsymbol{\zeta}^t$.
6: **end procedure**

Algorithm 3. Least squares policy iteration with RLSE.

1: Initialize the configuration s_0.
2: Initialize $\boldsymbol{\theta}^0 = \boldsymbol{\theta} = \mathbf{0}$ and $B^0 = \epsilon I$.
3: Initialize $\boldsymbol{\zeta}^0 = \mathbf{0}$ and $\overline{B}^0 = \epsilon I$.
4: **for** t=1,2,3,... **do**
5: Let $\hat{q}_t$ be the just received query.
6: $s_t \leftarrow \underset{s \in S_{s_{t-1}, \hat{q}_t}}{\arg\min} (\boldsymbol{\zeta}^{t-1})^T \boldsymbol{\eta}(s_{t-1}, q(s_{t-1}, s)) + (\boldsymbol{\zeta}^{t-1})^T \boldsymbol{\eta}(s, \hat{q}_t) + \gamma \boldsymbol{\theta}^T \boldsymbol{\phi}(s)$
7: Change the configuration to s_t.
8: Execute query $\hat{q}_t$.
9: $\hat{C}^t \leftarrow \delta(s_{t-1}, s_t) + cost(s_t, \hat{q}_t)$.
10: $\hat{\epsilon}^t \leftarrow (\boldsymbol{\zeta}^{t-1})^T \boldsymbol{\eta}(s_{t-1}, \hat{q}_t) - cost(s_{t-1}, \hat{q}_t)$
11: $B^t \leftarrow B^{t-1} - \frac{B^{t-1}\boldsymbol{\phi}(s_{t-1})(\boldsymbol{\phi}(s_{t-1}) - \gamma\boldsymbol{\phi}(s_t))^T B^{t-1}}{1 + (\boldsymbol{\phi}(s_{t-1}) - \gamma\boldsymbol{\phi}(s_t))^T B^{t-1}\boldsymbol{\phi}(s_{t-1})}$.
12: $\boldsymbol{\theta}^t \leftarrow \boldsymbol{\theta}^{t-1} + \frac{(\hat{C}^t - (\boldsymbol{\phi}(s_{t-1}) - \gamma\boldsymbol{\phi}(s_t)^T\boldsymbol{\theta}^{t-1})B^{t-1}\boldsymbol{\phi}(s_{t-1})}{1 + (\boldsymbol{\phi}(s_{t-1}) - \gamma\boldsymbol{\phi}(s_t))^T B^{t-1}\boldsymbol{\phi}(s_{t-1})}$.
13: $(\overline{B}^t, \boldsymbol{\zeta}^t) \leftarrow RLSE(\hat{\epsilon}^t, \overline{B}^{t-1}, \boldsymbol{\zeta}^{t-1}, \boldsymbol{\eta}^t)$
14: **if** ($\boldsymbol{\theta}^t$) converges **then**
15: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta}^t$.
16: **end if**
17: **end for**

Assume that there exists a feature mapping $\boldsymbol{\eta}$ such that $cost(s, q) \approx \boldsymbol{\zeta}^T \boldsymbol{\eta}(s, q)$ for some vector $\boldsymbol{\zeta}$. Changing the configuration from s to s' can be considered as executing a special query $q(s, s')$. Therefore we approximate

$$\delta(s, s') = cost(s, q(s, s')) \approx \boldsymbol{\zeta}^T \boldsymbol{\eta}(s, q(s, s')).$$

The vector $\boldsymbol{\zeta}$ can be updated iteratively using the well-known *recursive least squares estimation* (RLSE) [22] as shown in Algorithm 2, where $\boldsymbol{\eta}^t = \eta(s_{t-1}, \hat{q}_t)$ and $\hat{\epsilon}^t = (\boldsymbol{\zeta}^{t-1})^T \boldsymbol{\eta}^t - cost(s_{t-1}, \hat{q}_t)$ is the prediction error. Combining RLSE with LSPI, we get our cost-model oblivious algorithm as shown in Algorithm 3.

In Algorithm 3, the vector $\boldsymbol{\theta}$ determines the current policy. We can make decision by solving the equation in line 6. The values of $\delta(s_{t-1}, s)$ and $cost(s, \hat{q}_t)$ are obtained from the cost model. The vector $\boldsymbol{\theta}^t$ is used to approximate the cost-to-go function following the current policy. If $\boldsymbol{\theta}^t$ converges, then we update the current policy (line 14–16).

To check the efficiency and effectiveness of this algorithm, instead of using any heuristics we have initialzed policy π_0 as initial configuration s_0 and the cost-model C_0 as 0 shown in the lines 1–3 of Algorithm 3.

5 Case Study: Index Tuning

In this section, we present COREIL, an algorithm for tuning the configurations differing in their secondary indexes and handling the configuration changes corresponding to the creation and deletion of indexes, which instantiates Algorithm 3.

5.1 Reducing the Search Space

Let I be the set of indexes that can be created. Each configuration $s \in S$ is an element of the power set 2^I. For example, 7 attributes in a schema of R yield a total of 13699 indexes and a total of 2^{13699} possible configurations. Such a large search space invalidates a naive brute-force search for the optimal policy.

For any query $\hat{q}$, let $r(\hat{q})$ be a function that returns a set of recommended indexes. This function may be already provided by the database system (e.g., as with IBM DB2), or it can be implemented externally [1]. Let $d(\hat{q}) \subseteq I$ be the set of indexes being modified (update, insertion or deletion) by $\hat{q}$. We can define the reduced search space as

$$S_{s,\hat{q}} = \{s' \in S \mid (s - d(\hat{q})) \subseteq s' \subseteq (s \cup r(\hat{q}))\}. \tag{4}$$

Deleting indexes in $d(\hat{q})$ will reduce the index maintenance overhead and creating indexes in $r(q)$ will reduce the query execution cost. Note that the definition of $S_{s,\hat{q}}$ here is a subset of the one defined in Sect. 4.2 which deals with the general configurations.

Note that for tree-structured indexes (e.g., B^+-tree), we could further consider the *prefix closure* of indexes for optimization. For any configuration $s \in 2^I$, define the prefix closure of s as

$$\langle s \rangle = \{i \in I \mid i \text{ is a prefix of an index } j \text{ for some } j \in s\}. \tag{5}$$

Thus in Eq. (4), we use $\langle r(\hat{q}) \langle$ to replace $r(\hat{q})$ for better approximation. The intuition is that in case of $i \notin s$ but $i \subseteq \langle s \rangle$ we can leverage the prefix index to answer the query.

5.2 Defining the Feature Mapping ϕ

Let V be the cost-to-go function following a policy. As mentioned earlier, Algorithm 3 relies on a proper feature mapping $\boldsymbol{\phi}$ that approximates the cost-to-go function as $V(s) \approx \boldsymbol{\theta}^T \boldsymbol{\phi}(s)$ for some vector $\boldsymbol{\theta}$. The challenge lies in how to define $\boldsymbol{\phi}$ under the scenario of index tuning. In COREIL, we define it as

$$\phi_{s'}(s) := \begin{cases} 1, & \text{if } s' \subseteq s \\ -1, & \text{otherwise.} \end{cases}$$

for each $s, s' \in S$. Let $\boldsymbol{\phi} = (\phi_{s'})_{s' \in S}$. Note that $\phi_\emptyset$ is an intercept term since $\phi_\emptyset(s) = 1$ for all $s \in S$. The following proposition shows the effectiveness of $\boldsymbol{\phi}$ for capturing the values of the cost-to-go function V.

Proposition 2. *There exists a unique* $\boldsymbol{\theta} = (\theta_{s'})_{s' \in S}$ *which approximates the value function as*

$$V(s) = \sum_{s' \in S} \theta_{s'} \phi_{s'}(s) = \boldsymbol{\theta}^T \boldsymbol{\phi}(s). \tag{6}$$

Proof. Suppose $S = \{s^1, s^2, \ldots, s^{|S|}\}$. Note that we use superscripts to denote the ordering of elements in S.

Let $\boldsymbol{V} = (V(s))^T_{s \in S}$ and M be a $|S| \times |S|$ matrix such that

$$M_{i,j} = \phi_{s^j}(s^i).$$

Let $\boldsymbol{\theta}$ be a $|S|$-dimension column vector such that $M\boldsymbol{\theta} = \boldsymbol{V}$. If M is invertible then $\boldsymbol{\theta} = M^{-1}V$ and thus Eq. (6) holds.

We now show that M is invertible. Let $\boldsymbol{\psi}$ be a $|S| \times |S|$ matrix such that

$$\boldsymbol{\psi}_{i,j} = M_{i,j} + 1.$$

We claim that $\boldsymbol{\psi}$ is invertible and its inverse is the matrix $\boldsymbol{\tau}$ such that

$$\boldsymbol{\tau}_{i,j} = (-1)^{|s^i| - |s^j|} \boldsymbol{\psi}_{i,j}.$$

To see this, consider

$$\begin{aligned}(\boldsymbol{\tau}\boldsymbol{\psi})_{i,j} &= \sum_{1 \le k \le |S|} (-1)^{|s^i| - |s^k|} \boldsymbol{\psi}_{i,k} \boldsymbol{\psi}_{k,j} \\ &= \sum_{s_j \subseteq s_k \subseteq s_i} (-1)^{|s^i| - |s^k|}.\end{aligned}$$

Therefore $(\boldsymbol{\tau}\boldsymbol{\psi})_{i,j} = 1$ if and only if $i = j$. By the Sherman-Morrison formula, M is also invertible. □

However, for any configuration s, $\boldsymbol{\theta}(s)$ is a $|2^I|$-dimensional vector. To reduce the dimensionality, the cost-to-go function can be approximated by $V(s) \approx \sum_{s' \in S, |s'| \le N} \theta_{s'} \phi_{s'}(s)$ for some integer N. Here we assume that the collaborative benefit among indexes could be negligible if the number of indexes exceeds N. In particular when $N = 1$, we have

$$V(s) \approx \theta_0 + \sum_{i \in I} \theta_i \phi_i(s). \tag{7}$$

where we ignore all the collaborative benefits among indexes in a configuration. This is reasonable since any index in a database management system is often of individual contribution for answering queries [15]. Therefore, we derive $\boldsymbol{\phi}$ from Eq. (7) as $\boldsymbol{\phi}(s) = (1, (\phi_i(s))^T_{i \in I})^T$. By using this feature mapping $\boldsymbol{\phi}$, COREIL approximates the cost-to-go function $V(s) \approx \boldsymbol{\theta}^T \boldsymbol{\phi}(s)$ for some vector $\boldsymbol{\theta}$.

5.3 Defining the Feature Mapping η

A good feature mapping for approximating functions δ and *cost* must take into account both the benefit from the current configuration and the maintenance overhead of the configuration.

To capture the difference between the index set recommended by the database system and that of the current configuration, we define a function $\boldsymbol{\beta}(s, \hat{q}) = (1, (\boldsymbol{\beta}_i(s, \hat{q}))_{i \in I}^T)^T$, where

$$\boldsymbol{\beta}_i(s, \hat{q}) := \begin{cases} 0, & i \notin r(\hat{q}) \\ 1, & i \in r(\hat{q}) \text{ and } i \in s \\ -1, & i \in r(\hat{q}) \text{ and } i \notin s. \end{cases}$$

If the execution of query $\hat{q}$ cannot benefit from index i then $\boldsymbol{\beta}_i(s, \hat{q})$ always equals zero; otherwise, $\boldsymbol{\beta}_i(s, \hat{q})$ equals 1 or –1 depending on whether s contains i or not. For tree-structured indexes, we could further consider the prefix closure of indexes as defined in Eq. (5) for optimization.

On the other hand, to capture whether a query (update, insertion or deletion) modifies any index in the current configuration, we define a function $\boldsymbol{\alpha}(s, \hat{q}) = (\alpha_i(s, \hat{q}))_{i \in I}$ where

$$\alpha_i(s, \hat{q}) = \begin{cases} 1, & \text{if } i \in s \text{ and } \hat{q} \text{ modify } i \\ 0, & \text{otherwise.} \end{cases}$$

Note that if $\hat{q}$ is a selection query, $\boldsymbol{\alpha}$ trivially returns $\mathbf{0}$.

By combining $\boldsymbol{\beta}$ and $\boldsymbol{\alpha}$, we get the feature mapping $\boldsymbol{\eta} = (\boldsymbol{\beta}^T, \boldsymbol{\alpha}^T)^T$ used in COREIL. It can be used to approximate the functions δ and *cost* as described in Sect. 4.3.

6 Performance Evaluation

In this section, we present an empirical evaluation of COREIL. We implement a prototype of COREIL in Java and compare its performance with that of the state-of-the-art WFIT Algorithm [18]. WFIT is based on the Work Function Algorithm [5]. To determine the change of configuration, it considers all the queries seen so far and solves a deterministic problem towards minimizing the total processing cost.

6.1 Experimental Setup

We conduct all the experiments on a server running IBM DB2 10.5. The server is equipped with Intel i7-2600 Quad-Core @ 3.40 GHz and 4 GB RAM. We measure wall-clock times for execution of all components. Specially, for execution of workload queries or index creating/dropping, we measure the response time of processing corresponding SQL statement in DB2. Additionally, WFIT uses the what-if optimizer of DB2 to evaluate the cost. In this setup, each query is executed only once and all the queries were generated from one execution history. The scale factor (SF) used here is 2.

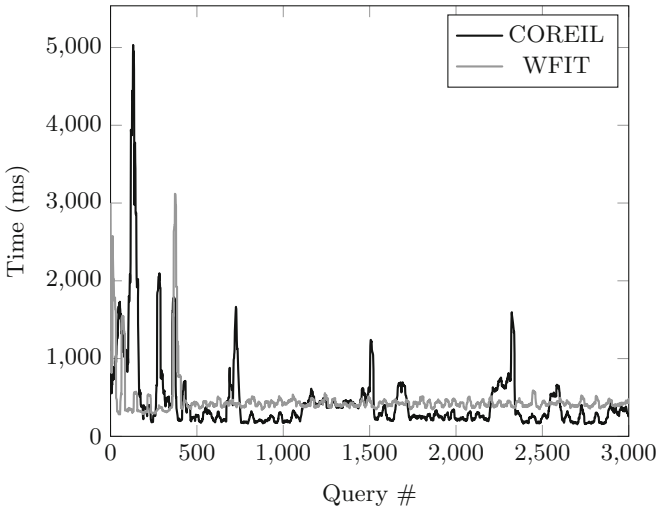

Fig. 1. Evolution of the efficiency (total time per query) of the two systems from the beginning of the workload (smoothed by averaging over a moving window of size 20)

6.2 Dataset and Workload

The dataset and workload is conforming to the TPC-C specification [14] and generated by the OLTP-Bench tool [8]. The 5 types of transactions in TPC-C are distributed as NewOrder 45 %, Payment 43 4 transactions are associated with $3 \sim 5$ SQL statements (query/update). Note that [18] additionally uses the dataset NREF in its experiments. However, this dataset and workload are not publicly available.

6.3 Efficiency

Figure 1 shows the total cost of processing TPC-C queries for online index tuning of COREIL and WFIT. Total cost consists of the overhead of corresponding tuning algorithm, cost of configuration change and that of query execution. Results show that, after convergence, COREIL has lower processing cost most of the time. But COREIL converges slower than WFIT, which is expected since it does not rely on the what-if optimizer to guide the index creations. With respect to the whole execution set, the average processing cost of COREIL (451 ms) is competitive to that of WFIT (452 ms). However, if we calculate the average processing cost of the 500^{th} query forwards, the average performance of COREIL (357 ms) outperforms that of WFIT (423 ms). To obtain further insight from these data, we study the distribution of the processing time per query, as shown in Fig. 2. As can be seen, although COREIL exhibits larger variance in the processing cost, its median is significantly lower that that of WFIT. All these results confirms that COREIL has better efficiency than WFIT under a long term execution.

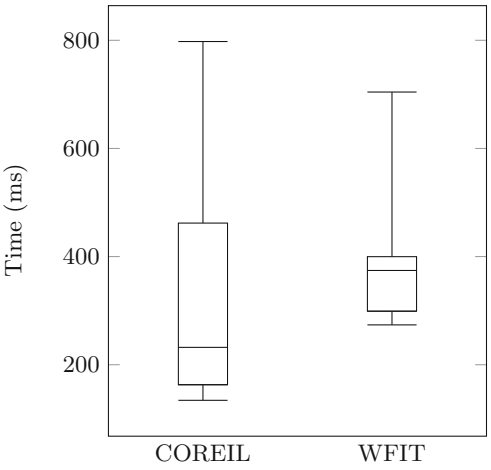

Fig. 2. Box chart of the efficiency (total time per query) of the two systems. We show in both cases the 9th and 91th percentiles (whiskers), first and third quartiles (box) and median (horizontal rule).

Figures 3 and 4 show analysis of the overhead of corresponding tuning algorithm and cost of configuration change respectively. By comparing Fig. 1 with Fig. 3, we can see that the overhead of the tuning algorithm dominates the total cost and the overhead of COREIL is significantly lower than that of WFIT. In addition, WFIT tends to make costlier configuration changes than COREIL, which is reflected in a higher time for configuration change. This would be discussed further in the micro-analysis. Note that both methods converge rather quickly and no configuration change happens beyond the 700th query.

6.4 Effectiveness

To verify the effectiveness of indexes created by the tuning algorithms, we extract the cost of query execution from the total cost. Figure 5 (note the logarithmic y-axis) indicates that the set of indexes created by COREIL shows competitive effectiveness with that created by WFIT, though WFIT is more effective in general and exhibits less variance after convergence. Again, this is to be expected since COREIL does not have access to any cost model for the queries. As previously noted, the total running time is lower for COREIL than WFIT, as overhead rather than query execution dominates running time for both systems.

We have also performed a micro-analysis to check whether the indexes created by the algorithms are reasonable. We observe that WFIT creates more indexes with longer compound attributes, whereas COREIL is more parsimonious in creating indexes. For instance, WFIT creates a 14-attribute index as shown below.

```
[S_W_ID, S_I_ID, S_DIST_10, S_DIST_09, S_DIST_08, S_DIST_07,
```

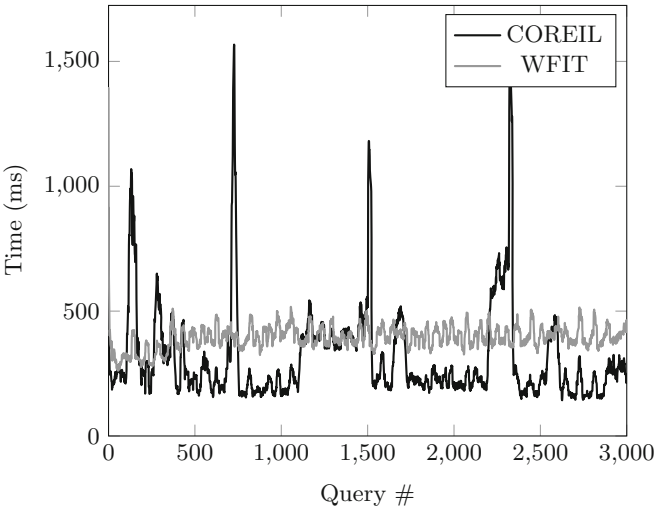

Fig. 3. Evolution of the overhead (time of the optimization itself) of the two systems from the beginning of the workload (smoothed by averaging over a moving window of size 20)

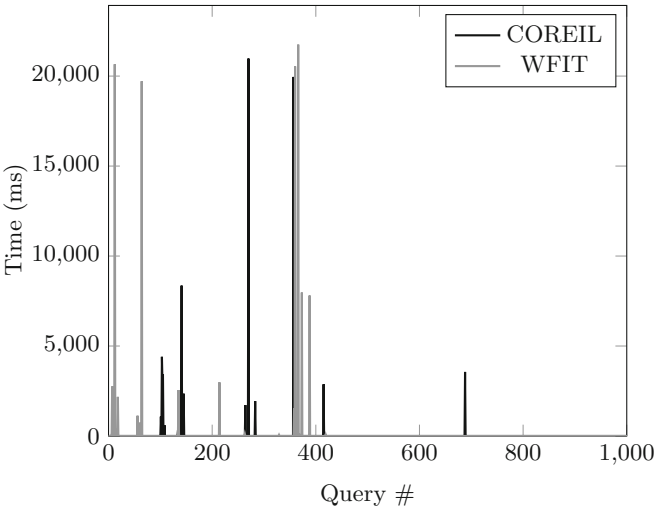

Fig. 4. Evolution of the time taken by configuration change (index creation and destruction) of the two systems from the beginning of the workload; no configuration change happens past query #1000

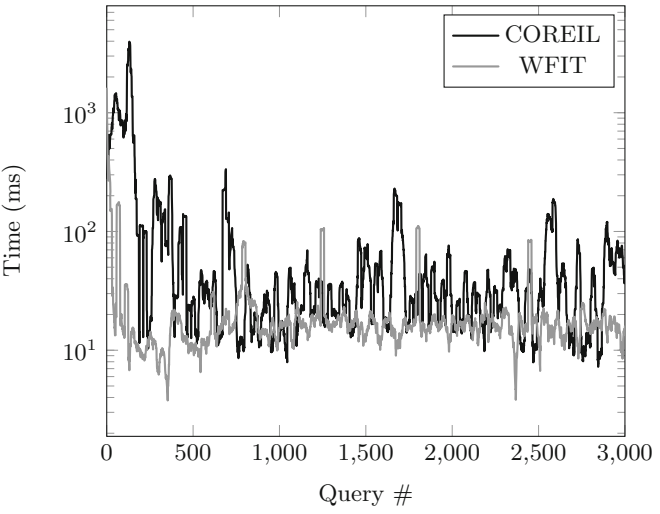

Fig. 5. Evolution of the effectiveness (query execution time in the DBMS alone) of the two systems from the beginning of the workload (smoothed by averaging over a moving window of size 20); logarithmic y-axis

```
 S_DIST_06, S_DIST_05, S_DIST_04, S_DIST_03, S_DIST_02,
 S_DIST_01, S_DATA, S_QUANTITY]
```

The reason of WFIT creating such a complex index is probably due to multiple queries with the following pattern.

```
SELECT S_QUANTITY, S_DATA, S_DIST_01, S_DIST_02, S_DIST_03,
       S_DIST_04, S_DIST_05, S_DIST_06, S_DIST_07, S_DIST_08,
       S_DIST_09, S_DIST_10
FROM STOCK
WHERE S_I_ID = 69082 AND S_W_ID = 1;
```

In contrast, COREIL tends to create shorter compound-attribute indexes. For example, COREIL created an index `[S_I_ID, S_W_ID]` which is definitely beneficial to answer the query above and is competitive in performance compared with the one created by WFIT.

7 Conclusion

We have presented a cost-model oblivious solution to the problem of performance tuning. We have first formalized this problem as a Markov decision process. We have devised and presented a solution, which addresses the curse of dimensionality. We have instantiated the problem to the case of index tuning and implemented the COREIL algorithm to solve it. Experiments show competitive performance with respect to the state-of-the-art WFIT algorithm, despite COREIL being cost-model oblivious.

Now that we have validated the possibility for cost-model oblivious database tuning, we intend in future work to study the trade-off for COREIL between efficiency and effectiveness in the case of index tuning. To show universality and robustness of COREIL, we are planning to run further tests on other datasets like TPC-E, TPC-H and benchmark for online index tuning. To find out its sensitivity on setup, we want to experiment with varying scale factors and and other parameters. Furthermore, we want to extend our approach to other aspects of database configuration, including partitioning and replication. This is not straightforward, as the solution will require heuristics that help curb the combinatorial explosion of the configuration space as well as may need some intelligent initialization technique.

Acknowledgement. This research is funded by the National Research Foundation Singapore under its Campus for Research Excellence and Technological Enterprise (CREATE) programme with the SP2 project of the Energy and Environmental Sustainability Solutions for Megacities - E2S2 programme.

References

1. Agrawal, S., Chaudhuri, S., Narasayya, V.R.: Automated selection of materialized views and indexes in SQL databases. In: VLDB (2000)
2. Agrawal, S., Narasayya, V., Yang, B.: Integrating vertical and horizontal partitioning into automated physical database design. In: SIGMOD (2004)
3. Azefack, S., Aouiche, K., Darmont, J.: Dynamic index selection in data warehouses. CoRR abs/0809.1965 (2008). http://arxiv.org/abs/0809.1965
4. Benedikt, M., Bohannon, P., Bruns, G.: Data cleaning for decision support. In: CleanDB (2006)
5. Borodin, A., El-Yaniv, R.: Online Computation and Competitive Analysis. Cambridge University Press, Cambridge (1998)
6. Bruno, N., Chaudhuri, S.: An online approach to physical design tuning. In: ICDE (2007)
7. Chaudhuri, S., Narasayya, V.: Autoadmin: What-if index analysis utility. In: SIGMOD (1998)
8. Difallah, D.E., Pavlo, A., Curino, C., Cudre-Mauroux, P.: OLTP-Bench: an extensible testbed for benchmarking relational databases. Proc. VLDB Endow. **7**(4), 277–288 (2013)
9. Lagoudakis, M.G., Parr, R.: Least-squares policy iteration. J. Mach. Learn. Res. **4**, 1107–1149 (2003)
10. Lohman, G.M.: Is query optimization a "solved" problem? (2014). http://wp.sigmod.org/?p=1075
11. Papadomanolakis, S., Dash, D., Ailamaki, A.: Efficient use of the query optimizer for automated physical design. In: VLDB (2007)
12. Powell, W.B.: Approximate Dynamic Programming: Solving the Curses of Dimensionality. Wiley-Interscience, New York (2007)
13. Puterman, M.L.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. Wiley, New York (2009)
14. Raab, F.: TPC-C - the standard benchmark for online transaction processing (OLTP). In: Gray, J. (ed.) The Benchmark Handbook. Morgan Kaufmann, San Francisco (1993)

15. Ramakrishnan, R., Gehrke, J., Gehrke, J.: Database Management Systems, vol. 3. McGraw-Hill, New York (2003)
16. Rasin, A., Zdonik, S.: An automatic physical design tool for clustered column-stores. In: EDBT (2013)
17. Schnaitter, K., Abiteboul, S., Milo, T., Polyzotis, N.: On-line index selection for shifting workloads. In: SMDB (2007)
18. Schnaitter, K., Polyzotis, N.: Semi-automatic index tuning: keeping DBAs in the loop. Proc. VLDB Endow. **5**(5), 478–489 (2012)
19. Stillger, M., Lohman, G.M., Markl, V., Kandil, M.: LEO - DB2's LEarning Optimizer. In: VLDB (2001)
20. Sutton, R.S., Barto, A.G.: Reinforcement Learning. MIT Press, Cambridge (1998)
21. White, D.J.: Markov Decision Processes. Wiley, New York (1993)
22. Young, P.: Recursive least squares estimation. In: Young, P. (ed.) Recursive Estimation and Time-Series Analysis, pp. 29–46. Springer, Berlin Heidelberg (2011)