Self-Driving DBs: Using Bandit Algorithms to Automate Index Selection in Database Systems

Shreyash Patodia
The University of Melbourne
spatodia@student.unimelb.edu.au

Abstract

Data is the single most important source of business intelligence for all organisations today. This has lead to Database Management Systems (DBMS) becoming very complex software systems with a large number of configuration knobs that can be used to optimise performance. In light of this, businesses traditionally hire experts known as Database Administrators (DBAs) to configure these knobs to achieve peak performance. However, as DBMS have become increasingly complex, it has become virtually impossible for humans to be able to optimally configure them. Vendors have thus started packaging DBMS with certain advisory tools that can help DBAs make configuration decisions. Ultimately however, the task of actual configuration is still the prerogative of the DBA since most of these advisory tools are far from perfect, owing to their heavy reliance on heuristic based methods and approximations.

Recent work in the area of machine learning has displayed its effectiveness in replacing human/heuristic based methods in a number of non-trivial tasks. In this paper we present a novel technique to automate index selection, which is one of the most critical configuration tasks performed by a DBA. We utilise a class of machine learning algorithms known as bandits and present a Knapsack Constrained Contextual Combinatorial Upper Confidence Bound Algorithm (C^3UCB) that extends the state-of-the-art in the area of bandit algorithms to tackle the problem of index selection. Our discussion in this report includes motivation, theoretical proofs and analytical results in order to justify our choice of algorithm and promote new work in this new and exciting area.

1 Introduction

The ability to collect, process, and analyze large amounts of data in order to infer business intelligence is extremely important for any organisation especially in this age of big data. Database Management Systems (DBMS) are a very important component of data-intensive ("Big Data") applications [1]. The performance of these systems is often measured in metrics such as throughput and latency. Achieving good performance in DBMS is a non-trivial task as they are extremely complex software systems with many tunable configurations that control nearly all aspects of their run-time operation [2]. Modern DBMS are notorious for having many knobs [3, 4, 5] the configuration of which completely determines the performance and scalability of the database. Furthermore, the default configurations of these knobs is infamously bad. For example, the default MySQL configuration in 2016 assumes that it is deployed on a machine that only has 160 MB of RAM [6]. Given this, many organisations resort to hiring expert Database Administrators (DBAs) to configure the system's knobs for the expected workload. These administrators try to optimise the database based on the current workload, statistics provided by the so-called design advisory tools, and through their past experiences/intuition to make the right decisions. As databases and applications grow in both size and complexity, optimising a DBMS to meet the needs of an application has surpassed the abilities of humans [7]. This is because

the correct configuration of a DBMS is highly dependent on a number of factors that are beyond what humans can reason about and this prompts the desire for automated methods that can effectively configure DBMS.

In this paper we explore the problem of automated index selection in a DBMS. A database index is an internal data structure used by the database to increase read speeds at the cost of write speeds. Index selection is a process which intends to produce efficient indices for a given query workload and is one of the most critical configurations made by the DBA to optimise database performance. As previously mentioned, DBAs currently use a combination of data, intuition, and experience in order to configure the database. In the last few years we've seen a flurry of non-trivial tasks that were previously performed by humans, heuristic based algorithms or a combination of the two be very successfully replaced by machine learning. The task of index selection lends itself perfectly to such a setting because we have all the data available to allow for a machine learning algorithm or a group of machine learning algorithms to replace a DBA. Specifically, we choose a class of machine learning algorithms known as bandit algorithms, as they tend to have a number of very desirable characteristics (which we will discuss in later sections) from the perspective of automated index selection.

Our contribution to the field of self-driving databases is in the form of the presentation of the novel idea of using bandit algorithms to automate database configuration, extending a current state-of-the-art bandit algorithm to apply it to the problem of automated index selection, provide theoretical guarantees about the performance of our algorithm, and develop an extendable platform for future development in the area and show experimental results that back up our theoretical claims analytically.

2 Background

2.1 Index Selection

A database index is an internal data structure used to increase read speeds at the cost of write speeds. It is simply a copy of selected columns of data from a table that can be searched very efficiently (due to ordering) and also includes a low-level disk block address or direct link to the complete row of data it was copied from. Indices can be made of one or more columns. An index can be represented in literature using an ordered tuple of columns from the same table in the database such as $(c_1, c_2, ...c_i)$ where i is the number of columns in the index. Moreover, index (c_1, c_2) is different from index (c_2, c_1) due to the relevance of ordering when it comes to indices. This means that a table with 5 columns can potentially have 5 single-columns indices, 20 two-column indices, 60 three-column indices, 120 four-column indices and another 120 five-column indices. We can clearly see how the number of possible indices in a database can grow exponentially as the number of tables and the number of columns in those tables increases. Moreover, index selection involves picking a subset of these indices for a given workload in order to optimise performance. If $\mathbb{I}$ represents the set of all indices in the database, $|\mathbb{I}|$ represents the number of possible of items in the set $\mathbb{I}$ then there are $2^{|\mathbb{I}|}$ subsets of indices that can theoretically be chosen. An exponential number of possibilities makes it virtually impossible for a human DBA or a traditional search algorithm to be able to choose the best set of indices for a non-trivial database. To add to the complexity of the problem, some other considerations that the DBA must make are: (1) any combination of indices we pick must be diverse enough to cover the variations in the workload of the database because picking very similar indices would mean that only a small number of the indices we create are getting used (2) the indices we pick will have to abide by any memory constraints of the machine that the database is on. These constraints, along with the exponential nature of the search space of the problem makes automated index selection an extremely difficult problem to solve and an extremely interesting area of research.

2.2 Multi-Armed Bandits

The multi-armed bandit (MAB) problem was first introduced by Robbins (1952) has been extensively studied in statistics and has been growing in popularity in the machine learning community over the last decade. A MAB problem can be formulated as a finite sequential decision-making problem, which lasts for n rounds. At the start of each of the n rounds a decision maker is presented with the choice of taking one of k actions (called the arms of the bandit), each having a reward distribution unknown to the decision maker. The goal of the decision maker is to maximise the total expected rewards over the course of n rounds. Every time the decision maker pulls an arm (chosen an action) they get a reward associated with the action of pulling the arm but receive no information about

the arms that were left untouched. This means that the decision maker must pull each arm enough times to be able to learn the distribution of the rewards for that arm. However, pulling an arm with a relatively low expected rewards too many times will lead to a drop in the cumulative rewards gathered by the decision maker. This leads to the exploration-exploitation dilemma that has been studied extensively in literature [8, 9, 10]. This problem for an algorithm to decide between exploring more of the problem space of exploiting what it already knows is one of the defining characteristics of the family of reinforcement learning algorithms that bandits can loosely be regarded a part of.

This simple bandit framework although extremely powerful does have its limitations, which has lead to extensions to the framework that allow for the usage of bandit algorithms in a number of scenarios. We utilise two such extensions in our work and the following paragraphs contain a short descriptions of each of the extensions:

- Contextual Bandits - The contextual bandit extends the model by making the decision conditional on the state of the environment. With such a model, we not only optimize decision based on previous observations, but also personalize decisions for every situation. For example, if a website wants to show users images they are likely to click on then the goal of the contextual bandit is to show an image of a cat to a cat person, and an image of a dog to a dog person, we may also show different images at different times of the day and days of the week. In a contextual bandit setting each arm is represented by a feature vector that is observed by the decision maker. Contextual bandits have recently been used to develop recommendation systems that adapt to user feedback. User feedback is an increasingly important source of data for online applications (e.g. Netflix project, Google news, Amazon) whose domains expand rapidly. New users don't have sufficient historical records, and hence are beyond the traditional recommendation technologies that anticipate a user's interest according to his/her past activities. A bandit algorithm helps decide which movie to recommend in the next round given the ratings in the previous rounds, i.e. whether we should try some new movies (exploration) or we should stick to the movies that the user has given high ratings to so far (exploitation). In particular, contextual bandits that model the reward as a linear function of context have proved to be extremely successful in practice [11] and we use a variation of this linear model in our work.
- Combinatorial Bandits - In a real-world scenario, we often want our bandit choose a set of actions to perform instead of choosing just one action. For example, recommender systems actually provide each user with a set of movies, rather than just one. In this setting, not one simple arm but a set of arms (generally called a super arm) are played together on each round. Such a setting is commonly called a Combinatorial Bandit setting [12]. The reward of a set of arms should not simply be the sum of ratings of each individual movie in this set. For example, we need a metric to qualify the diversity of the recommendation in the case of Netflix movie recommendations to avoid redundant or over-specified recommendation lists.

These two types of bandits fit well with the requirements of the index selection problem. The contextual bandit allows the decision maker to create indices based on incoming workload and the combinatorial bandit allows for the creation of multiple indices to be created at the start of each round. In order to impose memory limits we use per round knapsack constraints which makes sure that the total amount of memory the set of indices created at the start of each round occupies, is less than the memory constraint on the algorithm.

3 Related Work

Research in the area of autonomous DBMS dates back all the way to the 1970s with self-adaptive DBs [13]. The main focus of self-adaptive DBs was on the physical design of the DB. In fact, one of the first papers in this area also focused on automatic index selection [14]. The high-level ideas behind self-adaptive DBs were: (1) the system collects metrics on how the application accesses data and then (2) it searches for what change to make to improve performance based on a cost model. These ideas form the backbone of modern advisory tools but require the cost model to be extremely exhaustive in order to allow for good decision making. However, any model created using heuristic methods tend to have their drawbacks and these cost models are no different.

The 1990s saw the development of a number of database advisory tools that we continue to use to this day. At the forefront of this was work from Microsoft Research that helped the DBA select

the optimal indices [15], materialized views [16], and partitioning schemes [17] for their workload. Despite this, most of these tools only acted as an aid to the DBA and did not actually perform any actions themselves. Moreover, all of these works came with tools that were hard-coded and didn't learn from the actions they made in order to get better in their decision making. This means that even if an action that these tools recommended for a certain workload did not turn out to be a good action at all, they would keep on suggesting the same action for the same workload and not get better through the reinforcement the system would offer them in terms of the performance of the system.

Self-driving databases, which are the broad domain of this paper are much more recent. The term itself was coined by Pavlo in 2017 [18]. He listed that any database is self-driving if and only if it has:

- The ability to automatically select actions to improve some objective function (e.g., throughput, latency, cost). This selection also includes how many resources to use to apply an action.
- The ability to automatically choose when to apply an action.
- The ability to automatically learn from its actions and refine its decision making process.

As previously mentioned most of the work in the area of autonomous databases focuses on ability to select actions but none of the work prior to self-driving DBs actually tries to address the second or third requirements of a self-driving DB.

Related work in the area of automated index selection largely focuses on the use of heuristic-based methods to automate index selection or to guide database admins make to make better index choices [19, 20, 21, 22]. Jens Dittrich and his team at Saarland University have recently published work with a proof of concept for the use of Reinforcement Learning for automated index selection [23] that is the closest ideological equivalent to our work in this project. Nonetheless there is still one key difference: Their algorithm works in a supervised manner wherein it learns what indices are best for a given set of queries and is then used to suggest indices for test workloads. Our bandit on the other hand, is able to learn the relevance of specific indices to queries in an online manner and will constantly learn from experience to get better but this sort of behaviour is not going to be possible for the supervised method proposed by Dittrich et al. [23].

4 Our Approach

In this section, we formulate the problem of automated index selection as a multi-armed bandit problem, propose a variant of the Upper Confidence Bound (UCB) algorithm [24] which we call the Constrained Contextual Combinatorial Upper Confidence Bound (C^3UCB) algorithm, perform theoretical analysis to prove that the algorithm is expected to be near optimal with regards to the performance of bandit algorithms and also provide the results of experiments we performed on a Microsoft SQL Server Database.

4.1 Problem Formulation

Let n be the number of rounds for which we run our algorithm. This corresponds to the number of times we choose to create a new set of indices. Similarly, let k be number of arms of the bandit where each arm represents an index that can be created. We use the set $\mathbb{I}$ to be the set of all possible indices. At the start of each round, we receive the workload of the database as a series of queries and the task of our bandit algorithm is to choose a subset of indices that the algorithm expects will maximise the database performance whilst making sure that the indices that we create take less memory than a pre-specified limit M. This constraint M on the amount of memory is known as a knapsack constraint and we enforce it at the start of each round when we pick a subset of indices to create. Let $\mathbb{S} \subseteq 2^{[k]}$ be the set of all possible subsets of arms that use memory less than M. We call $S \in \mathbb{S}$ a *super arm* or *combinatorial arm*.

The learner observer k feature vectors $\{\mathbf{x}_t(1), ..., \mathbf{x}_t(k)\} \subseteq \mathbb{R}^d$ about the workload at the start of each round with each vector corresponding to an index in $\mathbb{I}$. Feature vector $\mathbf{x}_t(i)$ contain information about how relevant index i would be with respect to the queries in the workload. For example, how many times do the columns in the index appear in the queries in the workload and how many times the table the index belongs to appears in the queries in the workload. After observing these feature

vectors the learners chooses a super arm $S \in \mathbb{S}$ to play and observes a reward $R_t(S)$. $R_t(S)$ measures the quality of the super arm S and its definition will be specified later. The goal of the learner is to maximise the expected cumulative reward $\mathbb{E}[\sum_{t\in[n]} R_t(S)]$ over n rounds.

The reward $R_t(S)$ on round t is an application dependent function which measures the quality of recommended set of arms $S \subseteq [m]$. We choose the reward to be the gain the creation of this set of indices afforded us. Mathematically, our reward is: $R_t(S) = T_\emptyset(W) - T_S(W)$ where W is the workload and $T_S(W)$ is the time taken to execute the workload with an index set S. We divide this reward equally among all the indices that we chose to create. Since our current reward estimations might be a little bit simplistic we develop our framework to allow for different definitions of R without affecting any of our theoretical results. In general, we allow the expected reward $\mathbb{E}[R_t(S)]$ to be function of three variables: the super arm S, the feature vectors of the arms and the expected rewards from each arm which is described as $\theta_*^T \mathbf{x}_t(i)$ in the contextual bandit scenario. Formally, we denote the expected reward of playing S as $\mathbb{E}[R_t(S)] = f(S)$ and make the following assumptions about f:

- **Monotonicity** - The expected reward $f(S)$ is monotonically non-decreasing with respect to the score vector $\mathbf{r}$. Formally, for any set of feature vectors $\mathbf{X} = \{\mathbf{x}_t(1), ..., \mathbf{x}_t(k)\}$ of arms and super arm S, if $\mathbf{r}(i) \leq \mathbf{r}'(i)$ for all $i \in [m]$ we have $f_\mathbf{r}(S) \leq f_{\mathbf{r}'}(S)$
- **Lipschitz continuity** - The expected reward $f(S)$ is Lipschitz continuous with respect to the score vector r restricted on the arms in S. In particular, there exists a universal constant $C > 0$ such that, for any two score vectors $\mathbf{r}$ and $\mathbf{r}'$ we have $|f_\mathbf{r}(S) - f_{\mathbf{r}'}(S)| \leq C\sqrt{\sum_{i\in S} [\mathbf{r}(i) - \mathbf{r}'(i)]^2}$.

Given the expected scores $\mathbf{r}$ and arm vectors $\mathbf{X}$ as input, our program returns a prospective solution of the maximization problem $argmax_{S\in\mathbb{S}} f(S)$. Since the maximization problems of our reward function is NP-hard we produce what is called an α-approximation for some $\alpha \leq 1$ given input $\mathbf{r}$ and $\mathbf{X}$. The goal of the learner in this setting is to maximize its cumulative reward without knowing θ_* because the learner has no knowledge of the setting beforehand. With the knowledge of θ_*, the optimal strategy is to choose $S_* = argmax_{S\in\mathbb{S}} f(S)$ on round t. Hence, it is natural to evaluate a learner relative to this optimal strategy and the difference of the learner's total reward and the total reward of the optimal strategy is called regret. However, if a learner only has accesses to an α-approximation for some $\alpha \leq 1$, such evaluation would be unfair. Hence, we use the notion of α-regret which compares the learner's strategy with α-fraction of the optimal rewards on round t. Formally, the α-regret on round t can be written as

$$Reg_t^\alpha = \alpha opt - f(S)$$

and we are interested in designing an algorithm that minimises the α-regret.

4.2 Algorithm and Theoretical Analysis

In this section, we present the C^3UCB algorithm. The basic idea is to maintain a confidence set for the true parameter θ_*. For each round t, the confidence set is constructed from feature vectors $\mathbf{X}_1, ..., \mathbf{X}_{t-1}$ and observed scores of selected arms from previous rounds. We will show later that our construction of the confidence sets ensures that the true parameter θ_* lies in the confidence set with high probability. Using this confidence set of θ_* and feature vectors of arms $\mathbf{X}_t$, the algorithm can efficiently compute an upper confidence bound for the scores of each arm $\hat{\mathbf{r}}_t = \{\hat{r}_t(1), ..., \hat{r}_t(m)\}$. With the feature vectors and rewards as input, the algorithm returns a the super arm and uses the observed scores to adjust the confidence sets. The pseudo code of the algorithm is listed in Algorithm 30 where our notation is:

- $\mathbf{x}_n[i]$ is the context for index I_i at round n
- $p(q_n)$ is the set of columns involved in the predicate of the query at round n
- $m[i]$ is the memory required to build index I_i
- M is the size of the system's memory
- m is the size of the used memory. So, $M - m$ is the total of available space

- S_n is the set of arms that we are going to play at round n
- J is the number of tables in the database
- $\tau(I_i)$ returns the table which the index I_i performs at
- ι_n (iota) is the number of tables in which indexes are used at round n

Algorithm 1 Index Selection Based on C^{3}UCB

input: λ, α
$\mathbf{V}_0 \leftarrow \lambda \mathbf{I}_{K \times K}$
$\mathbf{b}_0 \leftarrow \mathbf{0}_K$
for $n = 1, \ldots, N$ **do**
  $\hat{\theta}_n \leftarrow \mathbf{V}_{n-1}^{-1} \mathbf{b}_{n-1}$
  for $i = 1, \ldots, K$ **do**
    $\mathbf{x}_n[i] \leftarrow \mathbf{0}_K$
    $(x_n[i])_i \leftarrow \mathbb{1}_{p(q_n)}(I_i)$ ▷ check whether the index can be used
    $\bar{r}_n[i] \leftarrow \hat{\theta}_t^T \mathbf{x}_n(i)$
    $\hat{r}_n[i] \leftarrow \bar{r}_n[i] + \alpha \sqrt{\mathbf{x}_n[i]^T \mathbf{V}_{n-1}^{-1} \mathbf{x}_n[i]}$ ▷ UCB for each index
  $s \leftarrow K$
  $S_n \leftarrow \phi$
  while $s > 0$ **do**
    for $i = 1, \ldots, K$ **do**
      if $m[i] > M - m$ and $\hat{r}_n[i] > -\infty$ **then**
        $\hat{r}_n[i] \leftarrow -\infty$
        $s \leftarrow s - 1$
    $i^\star \leftarrow \text{argmax}_i \frac{1}{m[i]} \hat{r}_n[i]$, prioritise $i^\star$ where $I_{i^\star} \in E_n$ in the case of tie break, choose uniformly at random if they have equivalent status
    $S_n \leftarrow S_n \cup \{I_{i^\star}\}$
    $m \leftarrow m + m[i^\star]$
    $\hat{r}_n[i^\star] \leftarrow -\infty$
    $s \leftarrow s - 1$
    for $i, \ldots, K$ **do**
      if i not $= i^\star$ and $prefix(I_i)$ = $prefix(I_{i^\star})$ **then**
        $\hat{r}_n[i] \leftarrow -\infty$
        $s \leftarrow s - 1$
  Play S_n and observe total rewards $R_n(S_n)$ as the performance gain
  $r[i] \leftarrow \frac{R_n(S_n)}{len(S_n)}$
  $\mathbf{V}_n \leftarrow \mathbf{V}_{n-1} + \sum_{i:I_i \in S_n} \mathbf{x}_n[i] \mathbf{x}_n[i]^T$
  $\mathbf{b}_n \leftarrow \mathbf{b}_{n-1} + \sum_{i:I_i \in S_n} r_n[i] \mathbf{x}_n[i]$

We assume that the $l2$-norms of parameter $\theta^\star$ and feature vectors of arms X_t are bounded to carry out our theoretical analysis. Using this assumption together with the monotonicity and Lipschitz continuity of the expected reward function f, the following theorem states that α-regret of the algorithm is bounded by $\tilde{\text{O}}(\sqrt{n})$ (where $\tilde{\text{O}}$ ignores all logarithmic terms).

THEOREM 1 (α-regret for Algorithm 1): Assuming $||\theta^\star||_2 \leq G$, $||\mathbf{x}_t(i)| \leq 1|$ and $r_t(i) \in [0, 1]$, for all $t \geq 0$ and $i \in [m]$. Given, $0 < \delta < 1$, set $\alpha_t = \sqrt{d \log(\frac{1 + tm/\lambda}{\delta})} + \lambda^{0.5} G$. Then, with probability $1 - \delta$, the total α-regret of C^3UCB algorithm satisties:

$$\sum_{t=1}^{n} Reg_t^\alpha \leq C \sqrt{64nd \log(1 + nm/d\lambda)}.(\sqrt{\lambda} G + \sqrt{2 \log(1/\delta) + d \log(1 + nm/(\lambda d))}$$

for any $n \geq 0$. The assumptions on $||\mathbf{x}_t(i)||_2 \leq 1$ and $r_t(i) \in [0, 1]$ can be satisfied by rescaling the values of the inputs accordingly.

To prove the theorem 1 we make use of some other theorems and lemmas. We start off by restating these theorems in our work:

THEOREM 2: suppose the observed scores $r_t(i)$ are bounded in $[0, 1]$. Assume that $||\theta_\star \leq G$ and $||\mathbf{x}_t(i)||_2 \leq 1$ for all $t \geq 0$ and $i \in [m]$. Define $\mathbf{V}_t = \mathbf{V} + \sum_{t=1}^{n} \sum_{i \in S} \mathbf{x}_t(i)\mathbf{x}_t(i)^T$ and set $\mathbf{V} = \lambda \mathbf{I}$. Thenn, with probability at least $1 - \delta$, for all rounds $t \geq 0$ the estimate $\hat{\theta}_t$ satisfies:

$$||\hat{\theta}_t - \theta_\star||_{\mathbf{V}_{t-1}} \leq \sqrt{d \log (\frac{1 + tm/\lambda}{\delta})} + \lambda^{0.5} G$$

This result essentially states that the true parameter $\theta_\star$ lies within the ellipsiod centered at $\hat{\theta}_t$ simultaneously for all $t \in [n]$ with high probability and was stated in [25].

Using this, theorem Qin et al. [26] proved the following lemma:

LEMMA 1: If we set $\alpha_t = \sqrt{d \log (\frac{1 + tm/\lambda}{\delta})} + \lambda^{0.5} G$, with the probability at least $1 - \delta$ we have:

$$0 \leq r_t\hat{(i)} - r_t^{\star}(i) \leq 2\alpha_t ||\mathbf{x}_t(i)||_{\mathbf{V}_{t-1}^{-1}}$$

holds simultaneously for any round $t \geq 0$ and any arm $i \in [m]$.

They also proved the following lemma:

LEMMA 2: Let $\mathbf{V} \in \mathbf{R}^{d \times d}$ be a positive definite matrix. For all $t = 1, 2, ...$, let S be a subset of $[m]$ of size less than or equal to k and define $\mathbf{V}_t = \mathbf{V} + \sum_{t=1}^{n} \sum_{i \in S} \mathbf{x}_t(i)\mathbf{x}_t(i)^T$. Then if $\lambda \geq k$ and $||\mathbf{x}_t(i)||_2 \leq 1$ for all t and i, we have:

$$\sum_{t=1}^{n} \sum_{i \in S} ||x_t(i)||^2_{\mathbf{V}_{t-1}^{-1}} \leq 2 \log (det \mathbf{V}_n) - \log (det \mathbf{V}) \leq 2d \log (trace(\mathbf{V})) + nk)/d) - 2 \log (det \mathbf{V})$$

With these two lemmas in place we are able to prove our THEOREM 1. Proof:

By LEMMA 1, we have $r^t(i) \geq r^t_{(i)}$ holds simultaneously for all $t[n]$ and $i[m]$ with probability at least 1δ. Now, assuming that this random event holds and applying the monotonicity property of the expected reward, for any super arm $S\mathbf{S}$, we have $f_{r^t}(S) \geq f_{r^t_\cdot}(S)$. We want to show that the super arm our algorithm picks will be $f(S) \geq \alpha opt$. Let $S^\star$ be super arm that actually maximises reward for a given workload and $\hat{S}$ be the arm that maximises the reward for our algorithm. We thus have:

$$\begin{aligned} f_{\hat{\mathbf{r}}_t}(S) &\geq \alpha opt_{\hat{\mathbf{r}}_t} \\ &= \alpha f_{\hat{r}_t}(\hat{S}) \\ &\geq f_{\hat{r}_t(S^*)} \\ &\geq f_{r_t^\cdot}(\hat{S^\star}) \\ &= \alpha opt_{\mathbf{r}_t^\star} \end{aligned}$$

Now we can bound α-regret at round t as follows:

$$
\begin{aligned}
Reg_t^{\alpha} &= \alpha opt_{\mathbf{r}_t^{\star}} - f_{r_t^{\star}}(S) \\
&\leq f_{\hat{r_t}}(S) - f_{r_t^{\star}}(S) \\
&\leq C\sqrt{\sum_{iinS}(\hat{r_t(i)} - r_t^{\star}(i))^2} \\
&\leq C\sqrt{\sum_{iinS}(4\alpha_t^2||\mathbf{x}_t(i)||^2_{\mathbf{V_{t-1}^{-1}}}}
\end{aligned}
$$

where the second inequality follows from Lipschitz continuity. Therefore, with probability at least $1-\delta$ for all $n \geq 0$,

$$
\begin{aligned}
\sum_{t=1}^{n} Reg_t^{\alpha} &\leq \sqrt{n\sum_{t=1}^{n}(Reg_t^{\alpha})^2} \\
&\leq C\sqrt{8n\sum_{t=1}^{n}\sum_{i\in S} 4\alpha_t^2||\mathbf{x}_t(i)||^2_{\mathbf{V}_{t-1}^{-1}}} \\
&\leq C\alpha_n\sqrt{32n}\sqrt{sum_{t=1}^{n}\sum_{i\in S} 4\alpha_t^2||\mathbf{x}_t(i)||^2_{\mathbf{V}_{t-1}^{-1}}} \\
&\leq C\alpha_n\sqrt{32n}\sqrt{2d\log(\lambda + nm/d) - 2d\log(\lambda)} \\
&\leq C\sqrt{64nd\log(\lambda+nm/d)}.(\sqrt{d\log((1+nm/\lambda)/\delta)} + \sqrt{\lambda}G)
\end{aligned}
$$

where the inequality follows from LEMMA 2, the fact that $|S \leq m$ for all t and that $\mathbf{V} = \lambda\mathbf{I}$. Upon simplification this gives us our proposed regret bound of $\tilde{\mathrm{O}}(\sqrt{n})$.

4.3 System Architecture

This project is one of the first steps we've taken towards building a self-driving database and at least two students are going to be working on this topic for the entirety of their doctoral studies. Thus, all of the development during the semester was done with a focus on setting up the foundations for a library that can be used to apply bandit algorithms to databases optimisation in a simple and extensible manner. Our current design can be used to test a number of different bandit algorithms against many different databases with relative ease due to the modular nature of the architecture and appropriate use of design patterns.

Our library runs as a thin layer on top of the database that is able to create appropriate indices by using the knowledge it has gained from past experiences in conjunction with information about the upcoming workload (represented by a series of queries) of the database. A concrete implementation of the *Bandit Policy* is given the workload of the DB as input, which it may choose to use as context. Each arm of the bandit corresponds to a possible DB index and is thus called an *Index Arm*. However, since we are dealing with a combinatorial bandit scenario where we need to choose more than one arm at the start of each round we created the concept of a *Combinatorial Arm* which is essentially a composition of a number of *Index Arms*. When a *Combinatorial Arm* of the bandit is pulled all the indices that make up the combinatorial arm are created and the workload is executed in order to receive the reward in terms of the time taken to execute it (the reward is of course conveyed to the bandit to be used as experience). The classes that implement the abstract *Database* class are used as adapters to allow easy communication with the underlying DBMS without having to change the APIs our Bandits interact with.

As we can see in Figure 1, the driver program uses the C^3UCB policy to decide which indices to create for the database. All the communication that the driver program or the combinatorial arm have

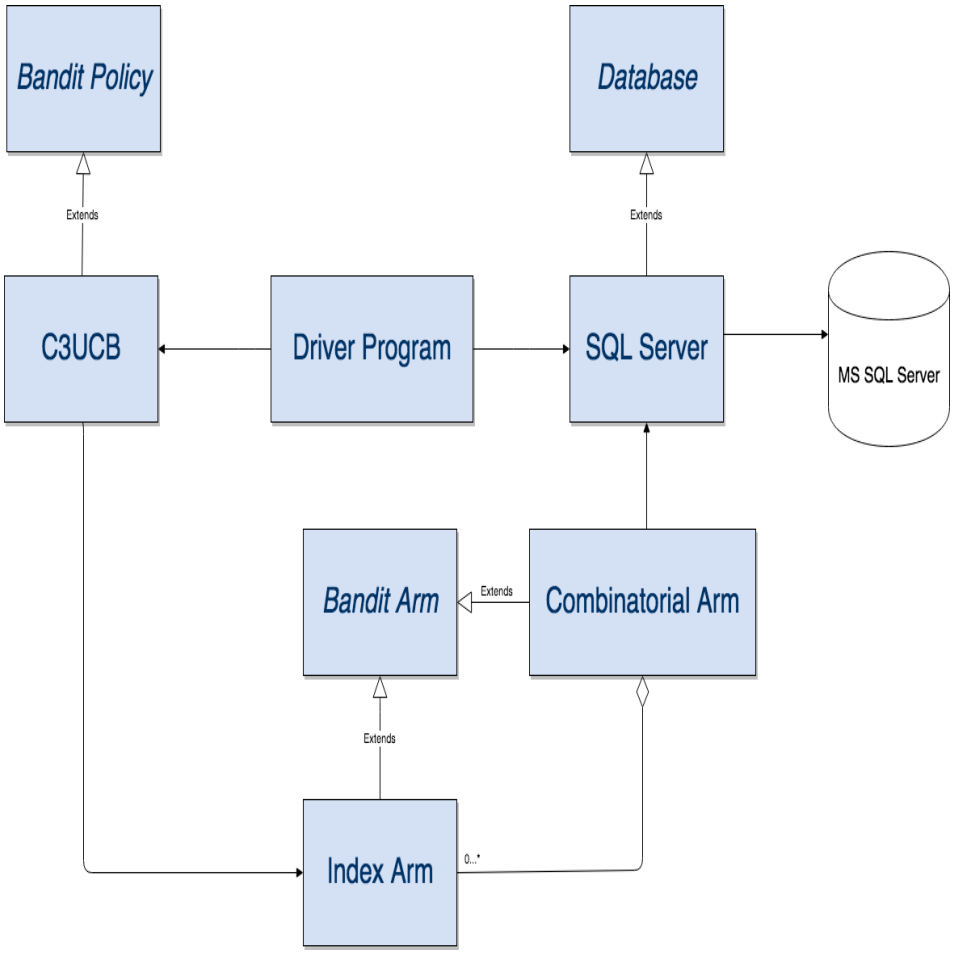

Figure 1: System Architecture

with the database goes through our *MS SQL Server* adapter class which means that if in the future we wanted to also test our algorithm on PostgreSQL we could simply write an adapter for it and switch out the MS SQL Server adapter for the PostgreSQL one. Similarly, if we wanted to test out other bandit algorithms, we could simply swap out C^3UCB for the new algorithm without having to modify the rest of the system too much.

We also provide machinery for the user to be able to do multiple runs of the same experiment and collect statistics. This is useful because the bandit setting is of a stochastic nature and the time taken to execute the same query can vary even though all other parameters in the experiment remain constant. We can keep on increasing the number of runs we average our results over to get closer and closer to the expected time taken to execute the query (this is in accordance with the Central Limit Theorem and Law of Large Numbers under the assumption that our observations for an arm are independent and identically distributed [27]).

4.4 Experimental Setup

To do our testing we chose the TPC-H benchmark [28], which is a decision support benchmark that consists of a suite of business-oriented ad hoc queries and concurrent data modifications. The queries and the data populating the TPC-H database have been chosen to have broad industry-wide relevance which makes them apt for us to simulate real-world data and workloads. Figure 2 shows an ER diagram of the design of the TPC-H database which we populated 1 GB of data (consisting of several million rows of information). We also added an additional table to our database to be able to store the

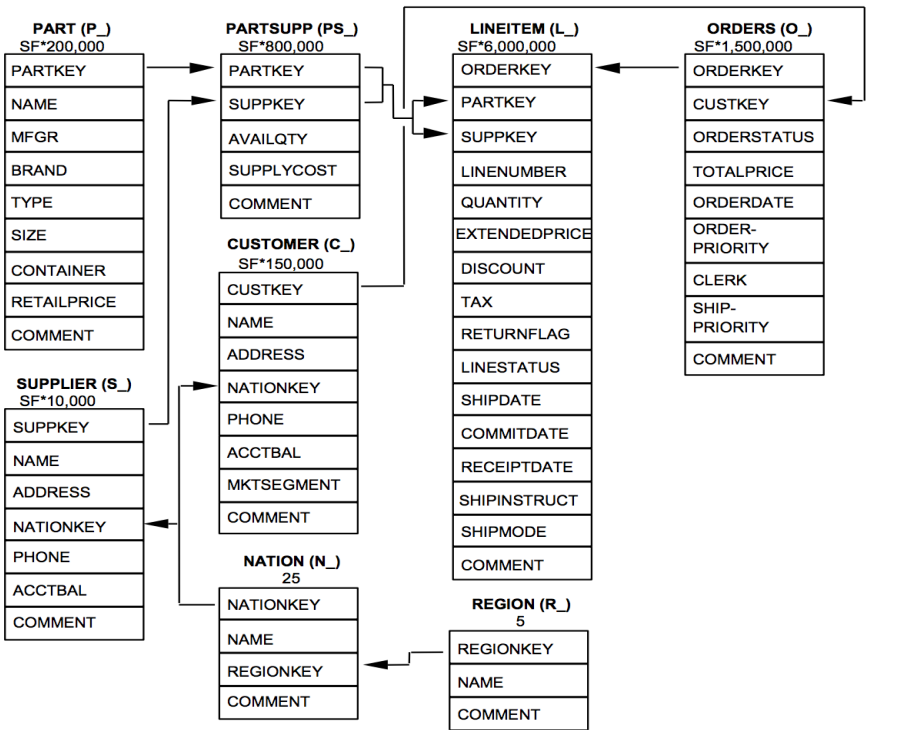

Figure 2: TPC-H Database

timings of all our queries. We generated 1,100 queries that were distributed uniformly among the 22 different families of queries in the TPC-H benchmark.

We ran most of our experiments with only one query in the workload to be able to get a granular picture of how well the algorithm is learning and performing for different queries. But we've also included a few experiments containing larger workloads that show that our algorithm should be robust to the size of the workload despite it potentially taking longer for it to learn due to the more varied nature of the workload. We also restricted the indices we used to single and double-column indices because the number of indices to deal with became exponentially large and it wasn't computationally possible to run experiments in a tractable manner with indices containing three or more columns. To put things into perspective there are a combined 581 one and two-column indices but there are over 5,000 three-column indices alone.

During the experiment, each time we chose a super arm for a given workload, We created all the indices in the super arm, executed the workload and then dropped the indices. This is analogous to DBAs creating a new set of indices during periods of lower traffic for the DB (for example early in the morning or late at night). We used the query plan to get the expected running time of a query devoid of any indices, $T_{\emptyset}(S)$. As mentioned during problem formulation, this value was used to measure gain in our reward function. Each experiment was run 10 times over and the result from each time-step were averaged out over the 10 runs. The memory constraint we used was a 100 MB of RAM, this number was arrived at after considering that the largest table in the database was 700 MB

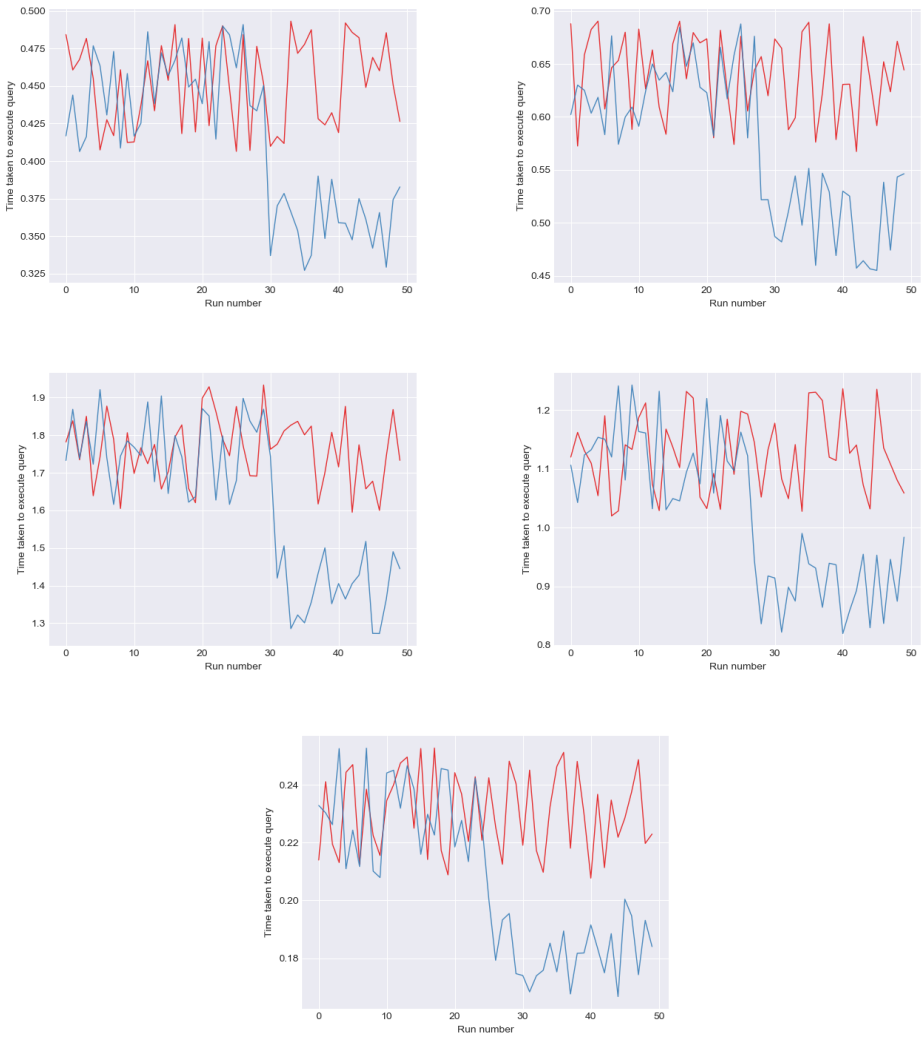

Figure 3: Results of running C^3UCB on 1 query per workload

and a two column index of this table will be around 88 MB in size. This allows for the possibility of creating two column indices from all the tables.

4.5 Experimental Results

We ran experiments with one query per workload over all of the 1,100 queries to see how well the algorithm learned to recognise the utility of specific indices for different queries. In Figure 3 shows the results of our testing for 5 different families of queries (1 query family per plot). Since there are 50 instances of queries from each family the x-axis represents the instance number, and the y-axis represents the time taken to run the query. The red lines represent running the query with no index whereas the blue lines represents running the query with indices created using our C^3UCB algorithm. There is a 10-20% decrease in the timings of the queries on average after 20-30 runs of the algorithm for that query. This performance is reflected throughout the other families of queries in the TPC-H benchmark. This performance bodes well towards the ability of our algorithm to be able to learn which indices to pick for a certain query.

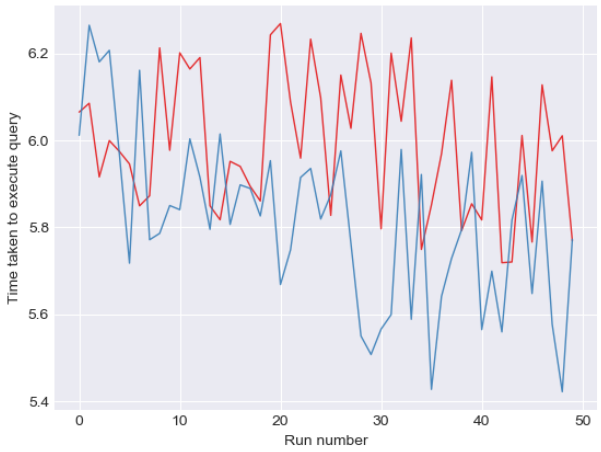

Figure 4: Results of running C^3UCB on 22 queries per workload

Figure 4 shows the performance of the algorithm when running all 22 types of queries per workload. We can see that the blue line where our algorithm creates the indices is able to give a 5% increase on average beyond round 25 of the algorithm. This might seem like a small amount but is quiet sizable considering the algorithm is only run for 50 rounds (it was not possible to run the algorithm for longer due to the time taken to run the current setup being several hours).

5 Conclusion

In this piece of work we've tried to show that Bandit Algorithms are an extremely good candidate to be used to do index selection for a database and have also motivated the ability of machine learning algorithms to be able to automate the tasks of a DBA. Our results on the TPC-H benchmark shows us that further work in the area of automating index selection using bandit algorithm could eventually perform just as well if not better than an human DBA.

6 Future Work

Since the field of self-driving databases is so new there are a number of directions in which future work could be directed. In the short term, work should be focused on alleviating some of the limitations of our current approach. For example, our approach currently assumes knowledge of the upcoming workload along with having ample past experiences to suggest a set of indices that might be optimal for the workload. However this might not always be representative of real life scenarios where the workload may not be known beforehand. It might thus be useful to explore using some subset of the past workloads to predict what future workloads are going to look like and create indices accordingly. If we could somehow predict future workloads effectively then it would be possible to use our current approach without having to worry about knowing upcoming workloads. Another limitation of our approach is that it takes some time for our algorithm to learn the relative usefulness of indices with respect to queries. This is because our algorithm is getting a cold-start and initially lacks any information about the relationships between the indices and queries. It might be possible to seed our algorithm with the decisions of the query optimiser so that we can somewhat avoid this cold-start problem. The algorithm can then learn to personalise index selection to the specific workloads that the database is being subject to instead of just suggesting the same indices that the query optimiser would.

In the longer term, we can look at applying machine learning techniques to other optimisation techniques used by database admins like the creation of materialised views or even normalising/denormalising of tables. Moreover, we could also look at the use of machine learning to optimise NoSQL Databases. Successful development of such techniques could lend help with the creation of

databases that can modify their own physical structure, tune knobs and holistically drive themselves better than any human database administrator ever could.

References

[1] Michael Stonebraker, Sam Madden, and Pradeep Dubey. Intel big data science and technology center vision and execution plan. *ACM SIGMOD Record*, 42(1):44–49, 2013.

[2] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. Tuning database configuration parameters with ituned. *Proceedings of the VLDB Endowment*, 2(1):1246–1257, 2009.

[3] Marc Linster. Best practices for becoming an exceptional postgres dba, 2014.

[4] Adam J Storm, Christian Garcia-Arellano, Sam S Lightstone, Yixin Diao, and Maheswaran Surendra. Adaptive self-tuning memory in db2. In *Proceedings of the 32nd international conference on Very large data bases*, pages 1081–1092. VLDB Endowment, 2006.

[5] Karl Dias, Mark Ramacher, Uri Shaft, Venkateshwaran Venkataramani, and Graham Wood. Automatic performance diagnosis and tuning in oracle. In *CIDR*, pages 84–94, 2005.

[6] MySQL – InnoDB startup options and system variables. `https://dev.mysql.com/doc/refman/5.6/en/innodb-parameters.html`. Accessed: 2018-10-26.

[7] Phil Bernstein, Michael Brodie, Stefano Ceri, David DeWitt, Mike Franklin, Hector Garcia-Molina, Jim Gray, Jerry Held, Joe Hellerstein, HV Jagadish, et al. The asilomar report on database research. *ACM Sigmod record*, 27(4):74–80, 1998.

[8] Mary J Benner and Michael L Tushman. Exploitation, exploration, and process management: The productivity dilemma revisited. *Academy of management review*, 28(2):238–256, 2003.

[9] Anil K Gupta, Ken G Smith, and Christina E Shalley. The interplay between exploration and exploitation. *Academy of management journal*, 49(4):693–706, 2006.

[10] Christian Stadler, Tazeeb Rajwani, and Florence Karaba. Solutions to the exploration/exploitation dilemma: Networks as a new level of analysis. *International Journal of Management Reviews*, 16(2):172–193, 2014.

[11] Lihong Li, Wei Chu, John Langford, and Robert E Schapire. A contextual-bandit approach to personalized news article recommendation. In *Proceedings of the 19th international conference on World wide web*, pages 661–670. ACM, 2010.

[12] Nicolo Cesa-Bianchi and Gábor Lugosi. Combinatorial bandits. *Journal of Computer and System Sciences*, 78(5):1404–1422, 2012.

[13] Michael Hammer. Self-adaptive automatic data base design. In *Proceedings of the June 13-16, 1977, National Computer Conference*, AFIPS '77, pages 123–129, New York, NY, USA, 1977. ACM.

[14] Michael Hammer and Arvola Chan. Index selection in a self-adaptive data base management system. In *Proceedings of the 1976 ACM SIGMOD International Conference on Management of Data*, SIGMOD '76, pages 1–8, New York, NY, USA, 1976. ACM.

[15] Surajit Chaudhuri and Vivek R. Narasayya. An efficient cost-driven index selection tool for microsoft sql server. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, VLDB '97, pages 146–155, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[16] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. Automated selection of materialized views and indexes in sql databases. In *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB '00, pages 496–505, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[17] Sanjay Agrawal, Vivek Narasayya, and Beverly Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 359–370, New York, NY, USA, 2004. ACM.

[18] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C Mowry, Matthew Perron, Ian Quah, et al. Self-driving database management systems. In *CIDR*, 2017.

[19] S. Chaudhuri, M. Datar, and V. Narasayya. Index selection for databases: a hardness study and a principled heuristic solution. *IEEE Transactions on Knowledge and Data Engineering*, 16(11):1313–1323, Nov 2004.

[20] Surajit Chaudhuri and Vivek R. Narasayya. An efficient cost-driven index selection tool for microsoft sql server. In *VLDB*, 1997.

[21] Sanjay Agrawal, Surajit Chaudhuri, and Vivek Narasayya. Automated selection of materialized views and indexes for sql databases.

[22] Sunil Choenni, Henk Blanken, and Thiel Chang. Index selection in relational databases. In *Computing and Information, 1993. Proceedings ICCI'93., Fifth International Conference on*, pages 491–496. IEEE, 1993.

[23] Ankur Sharma, Felix Martin Schuhknecht, and Jens Dittrich. The case for automatic database administration using deep reinforcement learning. *arXiv preprint arXiv:1801.05643*, 2018.

[24] Peter Auer. Using confidence bounds for exploitation-exploration trade-offs. *Journal of Machine Learning Research*, 3(Nov):397–422, 2002.

[25] Yasin Abbasi-Yadkori, Dávid Pál, and Csaba Szepesvári. Improved algorithms for linear stochastic bandits. In *Advances in Neural Information Processing Systems*, pages 2312–2320, 2011.

[26] Lijing Qin, Shouyuan Chen, and Xiaoyan Zhu. Contextual combinatorial bandit and its application on diversified online recommendation. In *Proceedings of the 2014 SIAM International Conference on Data Mining*, pages 461–469. SIAM, 2014.

[27] Anirban DasGupta. *Normal Approximations and the Central Limit Theorem*, pages 213–242. Springer New York, New York, NY, 2010.

[28] The TPC Benchmark H (TPC-H). `http://www.tpc.org/tpch/`.