Short Paper

Implementing Linear Bandits in Off-the-Shelf SQLite

Radu Ciucanu, Marta Soare, Sihem Amer-Yahia
{fname.lname}@univ-grenoble-alpes.fr
Univ. Grenoble Alpes, CNRS LIG

ABSTRACT

The *linear multi-armed bandit* is a reinforcement learning model that is largely used for sequential decision making in applications such as online advertising and recommender systems. We show that LinUCB, a well-known cumulative reward maximization algorithm for linear bandits, can be implemented in off-the-shelf SQLite. Additionally, our empirical study shows that, when dealing with small bandit data, our SQLite implementation is faster than an implementation in off-the-shelf Python. We believe that our findings open the door for many promising research directions on the topic of *in-DBMS federated learning* because (i) in the federated learning paradigm, many data owners contribute to the same learning task while locally storing their small data, and (ii) SQLite is a DBMS embedded in billions of devices, hence being able to implement federated learning on top of SQLite is of great practical interest.

1 INTRODUCTION

With the advent of machine learning (ML) and artificial intelligence (AI), and their use in a wide range of applications, there is also a growing interest within the database (DB) community to study what AI can do for DB and what DB can do for AI [14]. Regarding what DB can do for AI, there exists an important line of research on implementing linear algebra in the DBMS [16], which typically tackles the challenges of managing a large database.

In contrast, we focus here on a setting where we deal with implementing ML algorithms in an embedded DBMS, which deals with relatively small data. We are motivated by the emerging federated learning paradigm [11], where multiple data owners contribute to the same learning task, while keeping their small data stored locally and protected against privacy leaks. As mentioned in a state-of-the-art federated learning survey [11]: "*While numerous frameworks exist for data center training, the options for training models on resource constrained devices are fairly limited. Machine Learning models and training procedures are typically authored in a high level language such as Python [...] An ideal on-device runtime would have the following characteristics: Lightweight, Performant, Expressive [...] To our best knowledge no solution exists yet that satisfies these requirements, and we expect the limited ability to run ML training on end user devices to become a hindrance to adoption of federated technologies.*" Given the growing popularity of federated learning, we are interested in implementing ML algorithms in SQLite, which is[1] "*the most used database engine in the world [...] built into all mobile phones and most computers [...] bundled inside countless other applications that people use every day*". To the best of our knowledge, our work is the first that tackles the problem of implementing ML algorithms in SQLite and in embedded DBMS in general.

[1] https://www.sqlite.org/index.html

Within this research line, we focus on the reinforcement learning paradigm [19], in particular on the *linear multi-armed bandits* model [13, Ch. 19], largely used for sequential decision making in applications such as online advertising and recommender systems. We show it is possible to implement LinUCB, a well-known cumulative reward maximization algorithm for linear bandits, in off-the-shelf SQLite. We introduce the linear bandit model and the LinUCB algorithm in Section 2. Then, in Section 3, we outline our solutions for overcoming the challenges of the in-DBMS implementation of LinUCB, with no modifications to the DBMS, nor to the algorithm. Moreover, in Section 4, we present a preliminary empirical study comparing the clock time of our SQLite implementation to that of an implementation in off-the-shelf Python (using NumPy). Our in-DBMS LinUCB implementation with off-the-shelf SQLite is faster when dealing with datasets with few samples. In practice, such small datasets may be stored independently by several data owners, as motivated by federated learning settings. Finally, in Section 5 we discuss promising research directions for developing in-DBMS implementations of further federated reinforcement learning algorithms and settings.

2 PRIMER ON LINEAR BANDITS

The *stochastic multi-armed bandit* [13] is a sequential learning framework, which consists of a repeated interaction between a learner and the environment. The learner is given a set of choices (arms) with unknown associated rewards and a limited number of allowed interactions with the environment (budget). With the goal of maximizing the sum of the observed rewards, the learner sequentially chooses an arm and the environment responds with a stochastic reward corresponding to the chosen arm.

In the *linear stochastic multi-armed bandit* setting, the input set of arms is a fixed subset of $\mathbb{R}^d$, revealed to the learner at the beginning of the game. When pulling an arm, the learner observes a noisy reward whose expected value is the inner product between the chosen arm and an unknown parameter characterizing the underlying linear function (common to all arms).

Stochastic linear bandits can be used to model online recommendation: the arms are the objects that might be recommended and a reward is the user's response to a recommendation e.g., the click through rate or the rating associated to the recommendation. The recommender wants to maximize the sum of rewards, thus it needs to predict which object is more likely to be of interest for a certain user. The unknown parameter of the reward function is the user preference, more precisely the weights that the user gives to each of the d features in assessing an item.

In cumulative reward maximization algorithms, the learner faces the *exploration-exploitation dilemma*: at each round, decide whether to *explore* arms with more uncertain associated values, or *exploit* the information already acquired by selecting the arm with the seemingly largest value. Algorithms based on computing *upper confidence bounds (UCB)* on arm values are commonly used for cumulative reward maximization strategies [3, 4]. UCB-like algorithms guide the exploration-exploitation trade-off by updating, after each new observed reward, a *score* for each arm, given by the upper confidence bound of the estimated arm value.

Input: Budget N and K arms $x_1, x_2, \ldots, x_K$ in $\mathbb{R}^d$

Constants: Regularizer $\gamma > 0$; confidence parameter $\delta > 0$; noise parameter $R > 0$; $S > 0$ such that $||\theta||_2 \leq S$; $L > 0$ such that $\forall\, i \in [\![K]\!], ||x_i||_2 \leq L$

Unknown environment: Expected arm values; the learner has access only to the output of reward function $pull(x_i)$

Output: Sum of observed rewards for all arms

/ Initialization: Randomly pull an arm and initialize variables /

1 $r \leftarrow pull(x_i)$ */ Random reward (scalar) for arm x_i /*

2 $s \leftarrow r$ */ Sum of rewards of all arms (scalar) /*

3 $A \leftarrow \gamma I_d + x_i x_i^\top$ */ $(d \times d)$ matrix /*

4 $b \leftarrow r x_i$ */ $(d \times 1)$ vector /*

/ Exploration-Exploitation: At each round, pull an arm /

5 **for** $1 \leq t < N$

6 $\widehat{\theta} \leftarrow A^{-1} b$ */ Regularized least-squares estimate of θ /*

7 $\omega \leftarrow R\sqrt{d \cdot \log(\frac{1+tL^2/\gamma}{\delta})} + \gamma^{\frac{1}{2}} \cdot S$ */ Exploration term /*

/ For each arm i, based on current $\widehat{\theta}$, compute B_i, an UCB of $\langle x_i, \theta \rangle$. First term for exploitation, second for exploration /

8 **for** $1 \leq i \leq K$

9 $B_i \leftarrow \langle x_i, \widehat{\theta} \rangle + \omega ||x_i||_{A^{-1}}$

10 $x_m \leftarrow \arg\max_{i \in [\![K]\!]} B_i$ */ Ties broken at random /*

11 $r \leftarrow pull(x_m)$ */ Pull arm x_m /*

12 $s \leftarrow s + r$

13 $A \leftarrow A + x_m x_m^\top$

14 $b \leftarrow b + r x_m$

15 **return** s */ Return sum of observed rewards for all arms /*

Figure 1: LinUCB Algorithm [1].

As in a recent bandit textbook [13, Ch. 19], we use LinUCB as a generic name for UCB applied to stochastic linear bandits and we specifically rely on the algorithm of Abbasi-Yadkori et al. [1] in the case where the set of arms is fixed. In LinUCB (Figure 1), the arm scores are based on a regularized least-squares estimate of the unknown parameter of the linear reward function. At the next round, the arm with the largest updated score is pulled. We rely on the following notations:

- K is the number of arms; $[\![K]\!]$ is the set $\{1, 2, \ldots, K\}$.
- N is the budget = the number of observed rewards.
- d is the dimension of each vector: arms $x_1, \ldots, x_K$ and unknown parameter θ.
- x_i (for $i \in [\![K]\!]$) is an arm = a $(d \times 1)$ vector; we assume that all arms are pairwise distinct.
- θ is a $(d \times 1)$ vector (unknown to the learner) that is the parameter of the linear reward function.
- $||v||_2$ is the 2-norm of a $\mathbb{R}^d$ vector v.
- $||v||_A = \sqrt{v^\top A v}$ is the weighted 2-norm of a $\mathbb{R}^d$ vector v, where A is a $(d \times d)$ positive definite matrix.
- $\langle x_i, \theta \rangle$ (for $i \in [\![K]\!]$) is a scalar (unknown to the learner) defining the expected reward value of arm x_i, computed as the dot product of vectors x_i and θ.
- $pull(x_i)$ is a function that returns a noisy reward $\langle x_i, \theta \rangle + \eta$, where the noise η is e.g., drawn uniformly from the interval $[-R, R]$.

3 IMPLEMENTATION CHALLENGES

With a focus on the LinUCB algorithm introduced in Section 2, our first goal was to identify which of the needed computations can be easily done with existing techniques, and look for simple yet efficient SQL implementations for the others. In the rest of the section, we present the main ideas of our implementation. The most interesting technical parts are (i) the use of Sherman-Morrison formula [17] for computing the inverse of a matrix updated under certain constraints that are satisfied in LinUCB, and (ii) encoding the exploration-exploitation iterations of LinUCB using recursive views. The SQL code is available online[2].

3.1 Basic Data Structures and Computations

We store the set of arms as a matrix x, where each line i is the arm x_i. To encode matrices and basic computations, we use a standard technique [16] i.e., a matrix x is a SQL table with 3 columns (row id, column id, value). Encoding a vector (e.g., θ) is similar, the difference is that we need only one id. A common computation in LinUCB is matrix multiplication and we use a standard technique to encode it in SQL using a natural join, followed by group by and aggregation with sum on multiplication results. For example, the matrix multiplication result of two matrices `m(a,b,m_ab)` and `n(b,c,n_bc)` is:

```
select a, c, sum(m_ab * n_bc)
from m natural join n
group by a, c
```

Similar techniques work for other simple matrix computations needed by LinUCB: matrix-vector multiplication, inner (aka dot) product, or outer product.

3.2 Matrix Inverse

The trickiest matrix operation from LinUCB is matrix inverse, which is needed to update $\widehat{\theta}$ based on A^{-1} (line 6 in Figure 1). The challenge of implementing matrix inverses in-DBMS has been already acknowledged in the literature [16]. To the best of our knowledge, there does not exist yet an efficient technique that allows matrix inverse in-DBMS without using foreign function interfaces to other programming languages [16]. However, we observe that the matrix A that has to be inverted in LinUCB has a particular shape, in the sense that it is always updated by summing up with the outer product of a vector with itself i.e., $A \leftarrow A + x_m x_m^\top$ (line 13 in Figure 1). An existing technique from the AI/ML community [12, 21], but not yet used in the context of in-DBMS computations, rewrites the computation of an inverse of an updated matrix using basic operations (matrix difference and multiplication) on the inverse of the original matrix and the vectors used to update it. More precisely, the *Sherman-Morrison formula*[3] [17] is

$$(A + uv^\top)^{-1} = A^{-1} - \frac{A^{-1} u v^\top A^{-1}}{1 + v^\top A^{-1} u}$$

Consequently, in our implementation we decided to store not the matrix A, but rather A^{-1}. We compute the first A^{-1} i.e., $(\gamma I_d + x_i x_i^\top)^{-1}$ (line 3 in Figure 1) as part of the preprocessing and we give the first A^{-1} as SQL input to our LinUCB implementation similarly to how we specify all other algorithm input and constants (see[4] for an example). Then, in the LinUCB main code[2], we use the Sherman-Morrison formula (where both u and

[2] https://raw.githubusercontent.com/radu1/linucb-sqlite/main/linucb.sql
[3] https://en.wikipedia.org/wiki/Sherman-Morrison_formula
[4] https://raw.githubusercontent.com/radu1/linucb-sqlite/main/data.sql

v from the formula are x_m) to update A^{-1} based on standard SQL encoding of matrix multiplication and difference.

3.3 Simulating Iterations via Recursive Views

We recall (cf. Figure 1) that the budget N is spent during one initialization pull followed by $N-1$ exploration-exploitation pulls. Each of the $N-1$ iterations basically consists of pulling an arm (that is chosen as the argmax of the UCB score B_i) and then updating the different variables. To encode the iterations in SQL, we relied on a *recursive view* that iteratively inserts $N-1$ numbers in a table `iterations` that has a single column:

```
with recursive
  for(time_step) as (
    values(0)
    union all
    select time_step+1
    from for
    where time_step < (select N
                       from input)-2)
insert into iterations
  select (time_step)
  from for;
```

Moreover, we have a *trigger* that is activated after each insert in the table `iterations`:

```
create trigger if not exists update_explore_exploit
  after insert on iterations
begin
  -- pull arm with argmax
  insert into status values (
    1 + (select max(time_step)
         from status), -- t
    (select i
     from argmax_B_i), -- pulled arm
    (select(reward)
     from pull natural join argmax_B_i) -- reward
  );
  -- update A_inv
  [...]
  -- update b
  [...]
end;
```

The body of the trigger consists of SQL statements that update the different variables. In particular, for each of the two variables used to compute $\widehat{\theta}$ (i.e., matrix A^{-1} and vector b), we have a table in our SQL schema, and at each iteration we need the old version in order to compute the updated one. We created views for pulling an arm (including for generating random noise), computing $\widehat{\theta}$ and ω, and for computing arm scores B_i, as well as the argmax of these scores. The values of all these views are dynamically computed based on the different tables and/or other views.

4 EXPERIMENTS

We present a proof-of-concept experimental study, which shows the feasibility of our LinUCB implementation in off-the-shelf SQLite. We also identify cases when our implementation is faster than an implementation of LinUCB in Python.

To reproduce all our experimental findings, we make available our code and data on a public GitHub repository[5].

[5]https://github.com/radu1/linucb-sqlite

4.1 Competing Implementations

We compare the (clock) time of off-the-shelf single-threaded implementations of LinUCB in:

- SQLite[2], as outlined in Section 3;
- Python[6] i.e., a Python implementation of the pseudocode in Figure 1 using NumPy[7] [10] for linear algebra computations.

We wrote a preprocessing script[8] that compiles the linear bandit data (see Section 4.3) into the formats needed by the two languages e.g., SQLite[4] and Python[9]. In all reported experiments, the preprocessing time was in the order of milliseconds for both languages. The open source SQLite engine that runs our SQLite code has to be compiled with `SQLITE_ENABLE_MATH_FUNCTIONS` option (see our install script[10]) to be able to use math functions (e.g., log, sqrt). For a fair comparison, in the Python implementation we also used the Sherman-Morrison formula to rewrite the inverse computation of an updated matrix as a sequence of simpler matrix computations on the original inverse, which is faster that computing an inverse from scratch. After implementing LinUCB in the two languages, we performed a sanity check to be sure that both implementations yield the same correct arm selection strategy (i.e., same sequence of pulled arms), and output similar cumulative rewards. We did our experiments on a laptop with CPU Intel Core i7 of 2.80GHz and 16GB of RAM, running Ubuntu, and we report clock time averaged over 1000 runs.

4.2 Input Parameters and Constants

There are three parameters that dominate the computational complexity of LinUCB: budget N, number of arms K, and vector dimension d. We report results with $N \in \{10, 20, 30, 40\}$, $K \in \{10, 20, 30, 40\}$, and $d \in \{2, 4, 6, 8\}$. We rely on these numbers in order to show, for different combinations of parameters cf. Figure 2, the frontier between cases when SQLite is faster vs cases when Python is faster. Moreover, as shown in Figure 1, there are several LinUCB constants, that we fixed as inspired by other linear bandit papers [1, 5, 20]: $\gamma = 0.01$, $\delta = 0.001$, $R = 0.01$, $S = \log t$, and $L = \max ||x_i||_2$.

4.3 MovieLens Dataset

A popular motivation of both linear bandits [15] and federated learning [11] is in the domain of recommendation systems, hence we naturally relied on the classical *MovieLens 100K* dataset [9]. We build on a preprocessing result[11] [5] that we briefly recall next. The dataset is a collection of 100K movie ratings on a scale of 1 to 5, given by 943 users of the MovieLens website on 1682 movies. The collection of ratings is a matrix F (943×1682), whose element (i, j) is the rating of user i on movie j if the rating exists, or 0 otherwise. Since the user-movie matrix F is very sparse, it should be factored using low-rank matrix factorization. To this purpose, there already exists a Google Colab matrix factorization code[12], whose result is: a user embedding matrix U $(943 \times d)$, where row i is the embedding for user i, and a movie embedding matrix M $(1682 \times d)$, where row j is the embedding for movie

[6]https://raw.githubusercontent.com/radu1/linucb-sqlite/main/linucb.py
[7]https://numpy.org/
[8]https://raw.githubusercontent.com/radu1/linucb-sqlite/main/prep.py
[9]https://raw.githubusercontent.com/radu1/linucb-sqlite/main/data.py
[10]https://github.com/radu1/linucb-sqlite/blob/main/install.sh
[11]https://github.com/anatole33/LinUCB-secure/tree/master/extract_movie_lens
[12]https://github.com/google/eng-edu/blob/main/ml/recommendation-systems/recommendation-systems.ipynb

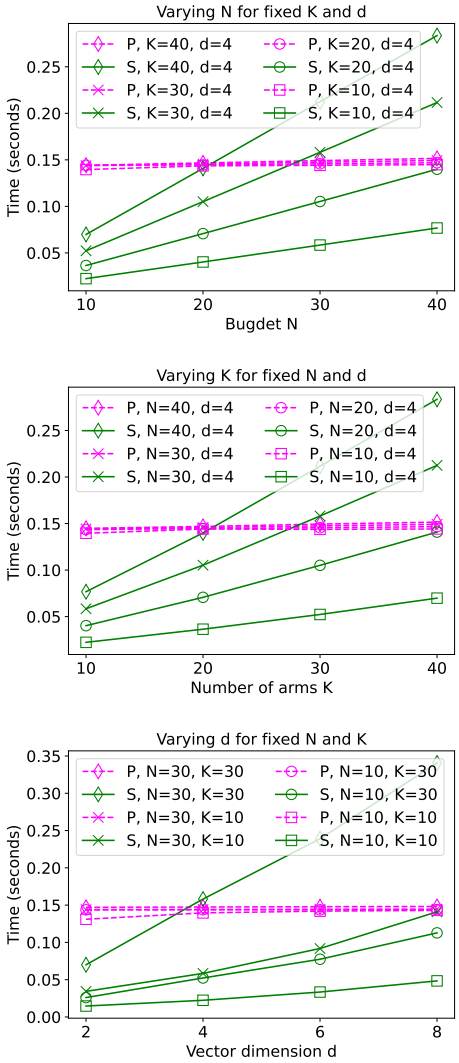

Figure 2: Comparison of LinUCB clock time in off-the-shelf SQLite (S) vs Python (P).

j. The embeddings are learned such that the product $UM^\top$ is a good approximation of the rating matrix F. The (i, j) entry of $UM^\top$ is the dot product of the embeddings of user i and movie j, computed such that it should be close to the (i, j) entry of F. Then, for every user i in matrix U, we can use linear bandit algorithms to recommend movies j from matrix M. In our experiments, the reported d values correspond to choices of d in the aforementioned matrix factorization approach, whereas the reported K arms correspond to choosing the embeddings of the first K movies in the dataset. As unknown environment θ, we choose the embedding of the first user i.e., we recommend movies and observe rewards based on the unknown intent of the first user in the dataset.

4.4 Results

We summarize our experimental results in Figure 2, in three different settings, where in each plot we vary one of the parameters N, K, d and we fix the other two. The first observation is that the behaviors of SQLite vs Python look linear vs constant, respectively. A second, more interesting observation is that Python is not always faster than SQLite. Indeed, we intuitively expected that an implementation in Python using the specialized linear algebra NumPy library should be always faster than an ad hoc implementation of linear algebra computations in an off-the-shelf DBMS engine. The aforementioned observations can be explained. Python has a constant overhead due to loading NumPy arrays, after which the increase in clock time correlated to the increase in parameter size is barely visible. On the other side, SQLite is very lightweight and particularly fast for small values of the three parameters, but it is not really optimized for linear algebra, hence its time increases much more visibly when the parameter size increases. Small data cases when SQLite is already better than Python (i.e., all green points under magenta points in Figure 2) could make sense in practice in federated learning settings. Hence, the arm vectors x_i and unknown environment vector θ may be distributed among many data owners, each of them storing small pieces of these vectors directly in the SQLite DBMS that is embedded in the data owners' devices.

5 FUTURE WORK

We believe that this paper opens the door for many promising research directions on the topics of linear algebra in embedded DBMS and on in-DBMS federated learning in general. Next, we discuss some interesting directions of future work.

First, we recall that our current SQLite implementation of LinUCB uses off-the-shelf SQLite and no particular optimization. It may be useful to investigate whether the current implementation can be improved either by finding more efficient techniques to encode the LinUCB computations in off-the-shelf SQLite, or by modifying the SQLite optimizer (that is open source) to tune it for the computations we need in LinUCB. Moreover, it would be interesting to see whether our implementation's techniques can be generalized to other linear bandit algorithms for cumulative reward maximization [2, 20] or for other optimization criteria [18], and more in general, to other bandit [13] and reinforcement learning [19] models. It would also be interesting to compare in-DBMS implementations as the one presented in this paper with ML systems e.g., SystemDS.

Our main motivation for implementing ML algorithms in an embedded DBMS is that such implementations could be useful in the context of federated learning [11], where multiple data owners store their raw data locally and participate to the learning task by sharing only limited pieces of data. Such an approach has the advantage of decreasing system latency by parallelizing computations between data owners, while also protecting their sensitive data. In particular, linear bandit algorithms often manipulate sensitive user data, given their use in online advertising and personalized recommendations. This raises the additional challenge of adapting privacy-preserving approaches for bandit algorithms (including cryptography [5–7] and differential privacy [8]) in order to design in-DBMS federated protocols that offer a good trade-off between data protection guarantees, computation time, and output accuracy.

ACKNOWLEDGMENTS

We thank the anonymous reviewers, whose suggestions helped us to improve our paper. This work has been partially supported by MIAI@Grenoble Alpes (ANR-19-P3IA-0003), and two projects funded by EU Horizon 2020 research and innovation programme (TAILOR under GA No 952215 and INODE under GA No 863410).

REFERENCES

[1] Y. Abbasi-Yadkori, D. Pál, and C. Szepesvári. 2011. Improved Algorithms for Linear Stochastic Bandits. In *Neural Information Processing Systems (NIPS)*. 2312–2320.

[2] M. Abeille and A. Lazaric. 2017. Linear Thompson Sampling Revisited. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*. 176–184.

[3] R. Agrawal. 1995. Sample Mean Based Index Policies with O(log n) Regret for the Multi-Armed Bandit Problem. *Advances in Applied Probability* 27, 4 (1995), pp. 1054–1078.

[4] P. Auer, N. Cesa-Bianchi, and P. Fischer. 2002. Finite-Time Analysis of the Multiarmed Bandit Problem. *Machine Learning* 47, 2-3 (2002), 235–256.

[5] R. Ciucanu, A. Delabrouille, P. Lafourcade, and M. Soare. 2020. Secure Cumulative Reward Maximization in Linear Stochastic Bandits. In *International Conference on Provable and Practical Security (ProvSec)*. 257–277.

[6] R. Ciucanu, P. Lafourcade, M. Lombard-Platet, and M. Soare. 2019. Secure Best Arm Identification in Multi-Armed Bandits. In *International Conference on Information Security Practice and Experience (ISPEC)*. 152–171.

[7] R. Ciucanu, P. Lafourcade, M. Lombard-Platet, and M. Soare. 2020. Secure Outsourcing of Multi-Armed Bandits. In *IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. 202–209.

[8] A. Dubey and A. Pentland. 2020. Differentially-Private Federated Linear Bandits. In *Neural Information Processing Systems (NeurIPS)*.

[9] F. M. Harper and J. A. Konstan. 2016. The MovieLens Datasets: History and Context. *ACM Transactions on Interactive Intelligent Systems (TiiS)* 5, 4 (2016), 19:1–19:19.

[10] C. Harris and et al. 2020. Array Programming with NumPy. *Nature* 585, 7825 (2020), 357–362.

[11] P. Kairouz and et al. 2021. Advances and Open Problems in Federated Learning. *Foundations and Trends in Machine Learning* 14, 1–2 (2021), 1–210.

[12] B. Kveton, C. Szepesvári, M. Ghavamzadeh, and C. Boutilier. 2019. Perturbed-History Exploration in Stochastic Linear Bandits. In *Conference on Uncertainty in Artificial Intelligence (UAI) (Proceedings of Machine Learning Research, Vol. 115)*. 530–540.

[13] T. Lattimore and C. Szepesvári. 2020. *Bandit Algorithms*. Cambridge University Press. https://tor-lattimore.com/downloads/book/book.pdf

[14] G. Li, X. Zhou, and L. Cao. 2021. AI Meets Database: AI4DB and DB4AI. In *International Conference on Management of Data (SIGMOD)*. 2859–2866.

[15] L. Li, W. Chu, J. Langford, and R. E. Schapire. 2010. A Contextual-Bandit Approach to Personalized News Article Recommendation. In *International Conference on World Wide Web (WWW)*. 661–670.

[16] S. Luo, Z. J. Gao, M. N. Gubanov, L. L. Perez, and C. M. Jermaine. 2019. Scalable Linear Algebra on a Relational Database System. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 31, 7 (2019), 1224–1238.

[17] J. Sherman and W. J. Morrison. 1949. Adjustment of an Inverse Matrix Corresponding to Changes in the Elements of a Given Column or a Given Row of the Original Matrix. *Annals of Mathematical Statistics* 20 (1949), 621.

[18] M. Soare, A. Lazaric, and R. Munos. 2014. Best-Arm Identification in Linear Bandits. In *Neural Information Processing Systems (NIPS)*. 828–836.

[19] R. S. Sutton and A. G. Barto. 2018. *Reinforcement Learning: An Introduction* (second ed.). The MIT Press. http://incompleteideas.net/book/the-book-2nd.html

[20] M. Valko, R. Munos, B. Kveton, and T. Kocák. 2014. Spectral Bandits for Smooth Graph Functions. In *International Conference on Machine Learning (ICML)*. 46–54.

[21] H. Wang, Q. Wu, and H. Wang. 2017. Factorization Bandits for Interactive Recommendation. In *AAAI Conference on Artificial Intelligence*. 2695–2702.