FeReD: Federated Reinforcement Learning in the DBMS

Sotirios Tzamaras
sotiris@tzamaras.com
Univ. Grenoble Alpes, CNRS LIG

Radu Ciucanu
radu.ciucanu@gmail.com
Univ. Grenoble Alpes, CNRS LIG

Marta Soare
marta.soare@univ-grenoble-alpes.fr
Univ. Grenoble Alpes, CNRS LIG

Sihem Amer-Yahia
sihem.amer-yahia@univ-grenoble-alpes.fr
CNRS LIG, Univ. Grenoble Alpes

ABSTRACT

Federated learning enables clients to enrich their locally trained models via updates performed by a coordination server based on aggregates of local models. There are multiple advances in methods and applications of federated learning, in particular in cross-device federation, where clients having limited data and computational resources collaborate in a joint learning problem. Given the constraint of limited resources in cross-device federation, we study the potential benefits of embedded in-DBMS learning, illustrated in a federated reinforcement learning problem. We demonstrate FeReD, a system that contrasts the performance of cross-device federation using Q-learning, a popular reinforcement learning algorithm. FeReD offers step-by-step guidance for in-DBMS SQLite implementation challenges for both horizontal and vertical data partitioning. FeReD also allows to contrast the Q-learning implementations in SQLite vs a standard Python implementation, by highlighting their learning performance, computational efficiency, succinctness and expressiveness. A video of FeReD is available at https://www.youtube.com/watch?v=2kRIu_C5RZA and its open source code at https://github.com/sotostzam/FeReD.

CCS CONCEPTS

• **Information systems** → **Federated databases**; • **Theory of computation** → **Reinforcement learning**; **Database query languages (principles)**.

KEYWORDS

In-DBMS reinforcement learning, Federated learning, SQLite

ACM Reference Format:
Sotirios Tzamaras, Radu Ciucanu, Marta Soare, and Sihem Amer-Yahia. 2022. FeReD: Federated Reinforcement Learning in the DBMS. In *Proceedings of the 31st ACM International Conference on Information and Knowledge Management (CIKM '22), October 17–21, 2022, Atlanta, GA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3511808.3557203

CIKM '22, October 17–21, 2022, Atlanta, GA, USA

ACM ISBN 978-1-4503-9236-5/22/10…$15.00
https://doi.org/10.1145/3511808.3557203

1 INTRODUCTION

Federated learning is an emerging paradigm that attracts a growing interest, both for fundamental developments and for a wide range of applications. One of its main advantages comes from allowing multiple clients (each having limited resources and data) to collaboratively solve a learning problem. Clients regularly transmit to a central coordination server their local models, and the server responds with an updated global model, computed as an aggregation of clients' models. Another advantage of federated learning is that clients only share local models, and not their raw data, thus avoiding some potential privacy leaks. The gain from federated learning is particularly visible in the *cross-device* federation setting that we consider in this work. As described in a recent survey [7], the cross-device federation has the following characteristics: the clients are mobile or IoT devices, each with limited data storage capacities and computational resources; each client stores its own data and cannot read the data of other clients; a central coordination server organizes the training, but never sees raw clients' data.

On the other hand, the same survey [7] points to an open problem concerning "*the limited options for training models on resource constrained devices*", which hinders a wider use of federated learning in practice. Moreover, the survey argues that "*an ideal on-device runtime would have the following characteristics: Lightweight, Performant, Expressive*". Motivated by this open problem and by recent advances on in-DBMS machine learning (ML) [4, 5, 9, 10], we study the gain obtained when each cross-device client performs the training directly in the embedded DBMS where the data is stored, rather than doing the training with a standard programming language.

We present FeReD, a system that allows to contrast the characteristics of performing local training with Python vs SQL implementations. We illustrate the features of FeReD in a *reinforcement learning* [13] setting, which can easily accommodate both vertical and horizontal data partitioning. We chose to focus on a reinforcement learning setting given the growth of the field, with an important number of recent theoretical advances [8], and real-world applications [6], including within federated protocols [12].

To the best of our knowledge, there is no prior work that studies the potential gain of doing local training by each client in their embedded DBMS in a reinforcement learning problem. The closest work to ours [5] shows that in a linear bandit algorithm, in-DBMS computations improve the performance when dealing with small data, which corresponds to the cross-device federation setting we consider in this paper. Their contributions include dealing with implementation challenges for translating iterative ML computations to SQL, but in a centralized setting. In contrast, here we investigate federated learning with both horizontal and vertical partitioning.

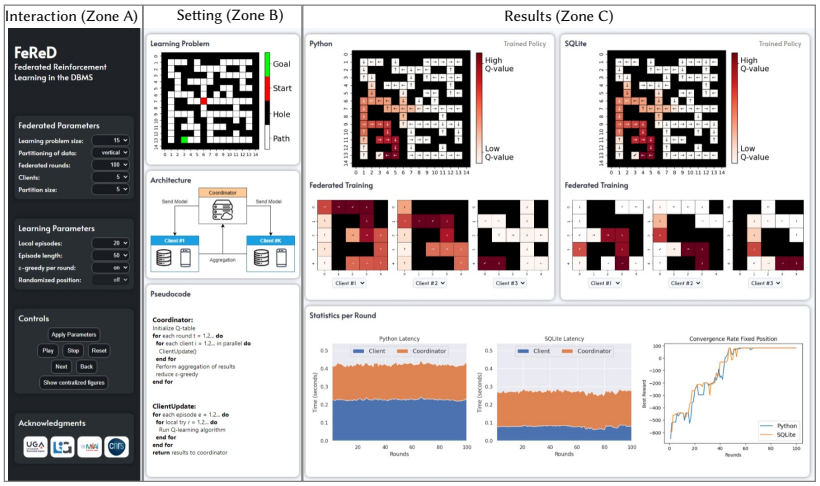

Figure 1: Interface of the FeReD system for in-DBMS federated reinforcement learning. Zone A is the interaction zone, where the data scientist configures the federated reinforcement learning protocol. Zone B presents the problem setting. Zone C depicts the results contrasting the performance of SQLite vs Python implementations on the client side.

Moreover, the learning problem we study here requires more complex computations, hence the implementation challenges we faced were also more complex. Additionally, FeReD is the first system that allows to study the gain of in-DBMS computations via a user-friendly interface and interactive displays of performance measures.

The demonstration targets data scientists that will use FeReD to configure a federated learning protocol and then evaluate several performance measures, including the gain in performance obtained by training clients' local models in SQL vs Python. The main finding is that **our SQL implementation has the same convergence rate to the best cumulative reward as a standard Python implementation, while being faster for both horizontal and vertical data partitioning in federated protocols.** One of the main technical contributions of FeReD is the translation of the Q-learning computations into SQL statements, which include the extensive use of recursive views and triggers. This points out the expressive power of SQL, which allows to encode complex reinforcement learning computations. To sum up, within FeReD we make the following contributions:

- We translate the computations done locally by each client into a SQL implementation. For this purpose, we rely on SQLite, "*the most used database engine in the world*" [2], which is already embedded into mobile phones and other typical devices used in cross-device federated learning settings.
- Via a user-friendly interface (see Fig. 1), FeReD allows data scientists to configure the federated protocol by choosing, for instance, the number of participating clients, and the type of data partitioning. FeReD supports both vertical and horizontal partitioning.
- For a chosen federated protocol, FeReD displays results for two equivalent federated implementations: the embedded in-DBMS SQLite implementation vs a standard Python implementation. The two implementations are contrasted with respect to performance measures such as: learning performance (that is, the convergence rate to the best cumulative reward), computational efficiency, succinctness and expressiveness.

Thus, FeReD helps data scientists to make an informed decision regarding the configuration of a federated reinforcement learning protocol. In the sequel, we present a system overview of FeReD (Section 2) and the demonstration scenarios (Section 3).

2 SYSTEM OVERVIEW

We introduce the considered reinforcement learning setting (Section 2.1), we discuss the technical challenges of implementing it as an in-DBMS federated learning protocol within FeReD (Section 2.2), and we present the features of FeReD's visual interface (Section 2.3).

2.1 Problem Setting

FeReD currently supports the classical *Q-learning* [14] algorithm, but its design and implementation techniques could be adapted to incorporate further reinforcement learning algorithms. The Q-learning algorithm learns the value of each action in a particular state through stochastic transitions, where "Q" refers to the quality function of any given state-action pair at time step t:

$Q^{new}(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha * (r_t + \gamma * \max Q(s_{t+1}, \alpha) - Q(s_t, a_t))$,

where r_t is the reward received by the agent in state s_t for action a_t, α is the learning rate, and $\gamma \in [0, 1]$ is the discount factor that determines how much the agent values rewards in the distant future (large γ value) relative to those in the immediate future ($\gamma = 0$). The algorithm uses a look-up Q-table storing the values for each state-action pair. A policy describes the agent's probability of choosing an action at any given time step. We follow the ϵ-decreasing strategy that chooses actions at random in the early stages of the algorithm (when ϵ is large), then focuses on actions with the highest estimated Q-value in the later stages (when ϵ is smaller). The Q-learning algorithm is implemented as an episodic process. Each episode has a fixed length, giving the number of actions the agent can take before the state of the environment resets and a new episode begins.

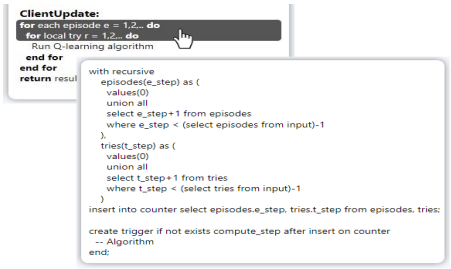

Figure 2: Snapshot of the pop-up window displaying the SQLite code selected in Zone B.

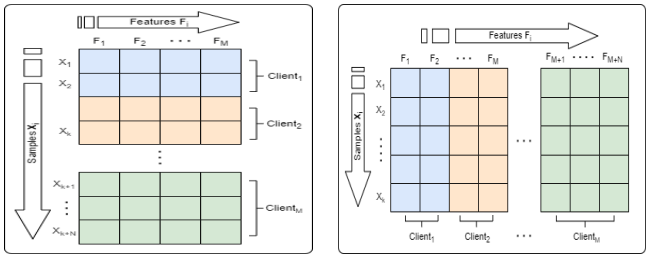

Figure 3: Horizontal (at the left) vs Vertical (at the right) data partitioning in a federated learning setting.

We consider the *Frozen Lake* [1] environment as the learning problem. In this environment (see an example on the top of Zone B in Fig. 1), an agent is asked to traverse a frozen lake from a starting position to the goal. The agent may not be able to move to a desired position, which indicates an unsafe state (aka a "hole"). The action space consists of four possible actions (go left/up/right/down) and there is a simple function which rewards the agent by a value of -1 for each successful action, a value of -10 for each action leading to an unsafe state and a reward of 100 for finding the goal.

2.2 Technical Challenges

Implementing Q-learning in SQLite. An interesting challenge was to **encode the iterations** of the Q-learning algorithm. To encode nested loops in SQLite, we used a **recursive view** (see Fig. 2). Given a fixed number of episodes and a fixed episode length, our recursive view iteratively inserts tuples (episode, current time step per episode) in a table `counter`. Moreover, a trigger is activated, with condition "`after insert`", which allows to perform complex SQL queries simulating the Q-learning algorithmic logic.

A second challenge was to implement a **breaking condition** for the inner loop, when the agent successfully finds the goal. Hence, the algorithm can advance to the next episode without executing unnecessary queries for the remaining actions of the current episode. Such a functionality is usually achieved by using a "flag" variable. Initially, we had the idea of using a `where` clause to check the value of a simple SQLite table, initialized with just one row and one column to act as a dynamic flag variable. This approach did not work because SQLite evaluates the `where` clause not for each iteration over the recursive view, but only once at the beginning. Our solution relies on three triggers with condition "`after insert`", which are responsible for three complementary tasks: (a) perform a Q-learning time step in the current episode, (b) reset the variables before doing a new time step, and (c) delete from the `counter` table the tuples for previous episodes and time steps.

Another important operation is to **generate random numbers** used for the ϵ-decreasing strategy. SQLite has a *random()* function that generates a random number in a predefined range [3]. Dividing the result by the maximum of this range allows number generation in the range $[0, 1]$. By rounding such a result in $[0, 1]$ and multiplying it by 3, we have the ability to pick an integer in the range $[0, 3]$, corresponding to the 4 actions that the agent has in our setting (that is, go left/up/right/down).

Implementing Q-learning in a federated setting. Another important challenge was to adapt the Q-learning implementations to a federated setting. We stress that the federated protocols implemented in FeReD cover the entire Q-learning algorithm, where learning (of the policy by the agent) and optimization (maximizing the cumulative reward) are done at the same time, as usually done in online ML. For completeness, FeReD supports both types of federated data partitioning [7] (see Fig. 3). For **horizontal partitioning**, the participating clients use Q-learning to learn a policy based on their own local copy of the global model (in our case the Q-table). In contrast to the centralized approach, in federated Q-learning, each client updates (in parallel) only a small fixed number of episodes, which implies runtime performance gains. For **vertical partitioning**, the clients' datasets are selected as small randomized subsets of the global model (in our case, subsets of the large global maze). The positions of the subsets are selected by the coordinating entity by starting around the goal state and working backwards towards exploring the best policy for the complete global model. When training is finished for all clients in a federated round, the resulting updates are transmitted to the coordinator. The coordinator aggregates these results to the global federated model using the *FederatedAveraging* [11] algorithm, slightly adapted for our Q-learning task. In particular, the coordinator averages the results of overlapping clients on the same state-action pair (cell of the maze).

2.3 Features of the Visual Interface

We present the visual interface of FeReD in Fig. 1 and we discuss next its main features.

Interaction (Zone A). This is where the data scientist can configure different parameters, more precisely:

- **Federated parameters.** Such parameters include the *number of clients* to be included in the federated learning protocol and the type of *data partitioning*, which can be horizontal or vertical. Both parameters will have a significant impact on the results displayed in the next zones. By selecting the horizontal partitioning, the clients receive a local copy of the global model to locally train. Instead, the selection of the vertical partitioning assigns a subset of the global model to each client. In addition, for the vertical partitioning, the data scientist can also select the partition size.
- **Learning parameters.** Such parameters will influence the difficulty of the learning problem, via the selected number of Q-learning episodes, their respective episode length, as well as the position of the starting point with respect to the goal.

- **Controls.** The data scientist applies the aforementioned selected parameters and can also choose to observe the results in a step-by-step manner (manually switching between federated rounds). Additionally, the data scientist can display in a pop-up window figures depicting offline comparisons contrasting the SQLite vs Python implementations in a standard centralized setting, with several combinations of parameters. These comparisons allow the data scientist to tune the federated parameters.

Setting (Zone B). This zone depicts the selected setting as configured by the data scientist. We first see the learning problem (that is, the maze for which we compute an efficient Q-learning policy). As typical for a frozen lake problem, the goal, the holes, and the starting point are known in advance. The goal of the agent is to learn a policy that assigns to each state-action pair values that would guide the agent into following the shortest path to the goal. In the middle of Zone B we display the federated architecture. We end Zone B with the pseudocode corresponding to global computations done by the coordinator, respectively to local computations that each client performs in parallel. By selecting a line in the pseudocode, the data scientist activates a pop-up showing the corresponding code for the SQLite implementation (see an example in Fig. 2).

Results (Zone C). In this zone, FeReD compares Python vs SQLite implementations according to several performance measures. On top of Zone C, we show the **global trained policy** corresponding to the entire federated learning process in each implementation. The shade in each cell corresponds to their Q-value as learned by each implementation. The darker the shade, the higher the Q-value of the best action (shown by the arrow) of the cell. Below, the data scientist can zoom on the maze (and its learned Q-values) for up to three selected clients, in each of the two implementations. We display the mazes of up to six clients at once such that visualization remains comfortable in FeReD's interface. Naturally, the individual performance of a client (i.e. its trained policy) is lower than that of the overall federated learning process, since we display what the client can learn only based on its local data.

Then, FeReD provides statistics per round. In the bottom-right of Zone C, FeReD displays the **convergence rate** of the two implementations. We notice that both implementations behave similarly in the attainable best cumulative rewards per round, and converge after the same number of federated rounds. FeReD also displays the time needed for the clients and the coordinator in the two implementations. The **latency time** for a federated round is the sum of: (i) the time taken by the slowest client, assuming that all clients work in parallel, and (ii) the time taken by the coordinator for the synchronization, which in our case means reading the messages received by each client, doing *FederatedAveraging* on their local data, and sending back the updates to all clients. In the plots, we report the sum of latency times for the entire federated protocol. The SQLite implementation is faster for both types of partitioning:

- For vertical partitioning, the gain is higher at every round, in particular when clients hold small subsets of the global model, run for fewer episodes and low values for episode lengths.
- For horizontal partitioning, the gain is higher after the first rounds, in particular when the value of ϵ decreases, and the agent starts to follow most of the times the same (optimal) path.

3 DEMONSTRATION SCENARIOS

The attendees will play the role of a data scientist who seeks to gather information about the best configuration for a federated reinforcement learning protocol. The attendees will simulate entire workflows of FeReD, configuring the learning problem and observing the different performance measures contrasting the SQLite and Python implementations on the clients' side. We consider two complementary demonstration scenarios.

Examining types of data partitioning. Our first scenario highlights the differences between SQLite vs Python implementations in **vertical vs horizontal data partitioning** scenarios, each contrasting the learning performance and the time required for each client to perform local computations. The difference between the two types of data partitioning will be visible in Zone C (see Fig. 1), where attendees will see what is the input maze (respectively, subset of the maze) solved by each client. Also in the upper part of Zone C, attendees will be able to compare the policy trained by each implementation, and in the lower part of Zone C, attendees will see statistics per round for the latency of the two implementations and their convergence rate. Notably, the vertical partitioning allows for a higher scalability. Indeed, in cross-device federated learning, one of the bottlenecks is the small amount of resources available in each client. By construction, in the horizontal partitioning each client has a local copy of a global maze, whereas in the vertical partitioning, each client has a subset of the global maze, the size of which can be chosen by the attendees. Hence, the vertical partitioning allows to scale to arbitrarily high values of global maze's size just by considering more clients among which to share (small) subsets covering the global maze. For example, according to our experimental results that we will share with the attendees, if every client has a 5×5 local maze, then FeReD works (with the SQLite implementation having a better running time performance than the Python one) for mazes of size $10K \times 10K$, which imply 4M clients. Such a number of clients is coherent with standard cross-device federated learning assumptions [7].

Examining SQL expressiveness. Our second scenario is focused on the comparison of the **expressiveness and succinctness** of the two equivalent implementations on the client side (local computations) in federated Q-learning. We will use a step-by-step comparison of Python vs SQLite starting from the pseudocode in the bottom of Zone B (see Fig. 1 and 2). Even if a standard Python implementation is clearly more succinct, one of the main findings enabled by the use of FeReD is that SQLite has an important expressive power allowing an equivalent federated implementation. We will also detail the challenges we faced for the SQLite implementation of Q-learning and our solutions, for instance the use of recursive views and the adaptation to federated protocols with both horizontal and vertical partitioning (see Section 2.2).

ACKNOWLEDGMENTS

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] Accessed in June 2022. Frozen Lake. https://www.gymlibrary.ml/environments/toy_text/frozen_lake/.

[2] Accessed in June 2022. SQLite. https://www.sqlite.org/index.html.

[3] Accessed in June 2022. SQLite Random Function. https://www.sqlitetutorial.net/sqlite-functions/sqlite-random/.

[4] M. Boehm, M. Dusenberry, D. Eriksson, A. Evfimievski, F. Manshadi, N. Pansare, B. Reinwald, F. Reiss, P. Sen, A. Surve, and S. Tatikonda. 2016. SystemML: Declarative Machine Learning on Spark. *Proc. VLDB Endow.* 9, 13 (2016), 1425–1436.

[5] R. Ciucanu, M. Soare, and S. Amer-Yahia. 2022. Implementing Linear Bandits in Off-the-Shelf SQLite. In *International Conference on Extending Database Technology (EDBT)*. 2:388–2:392.

[6] P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger. 2018. Deep Reinforcement Learning That Matters. In *AAAI Conference on Artificial Intelligence*. 3207–3214.

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

[8] V. Lavet, P. Henderson, I. Riashat, M. Bellemare, and J. Pineau. 2018. An Introduction to Deep Reinforcement Learning. *Foundations and Trends in Machine Learning* 11, 3-4 (2018), 219–354.

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

[10] 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.

[11] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. Agüera y Arcas. 2017. Communication-Efficient Learning of Deep Networks from Decentralized Data. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*. 1273–1282.

[12] J. Qi, Q. Zhou, L. Lei, and K. Zheng. 2021. Federated Reinforcement Learning: Techniques, Applications, and Open Challenges. *CoRR* abs/2108.11887 (2021). https://arxiv.org/abs/2108.11887

[13] 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

[14] C. Watkins and P. Dayan. 1992. Q-learning. *Machine Learning* 8, 3 (1992), 279–292.