ABSTRACT
Modern search and recommendation systems are optimized using logged interaction data. There is increasing societal pressure to enable users of such systems to have some of their data deleted from those systems. This paper focuses on “unlearning” such user data from neighborhood-based recommendation models on sparse, high-dimensional datasets. We present caboose, a custom top-k index for such models, which enables fast and exact deletion of user interactions. We experimentally find that caboose provides competitive index building times, makes sub-second unlearning possible (even for a large index built from one million users and 256 million interactions), and, when integrated into three state-of-the-art next-basket recommendation models, allows users to effectively adjust their predictions to remove sensitive items.

CSCS CONCEPTS
• Information systems → Data management systems; Recommender systems.

KEYWORDS
Machine unlearning, right-to-be-forgotten, nearest neighbors

1 INTRODUCTION
Ranking methods optimize search and recommendation systems so that the resulting rankings perform well for a given metric. Traditionally, most ranking methods applied a supervised learning procedure based on manually-created judgments. As an alternative, ranking methods have been developed that rely on logged user interactions [17].

The “right to be forgotten”. Recent law such as the “right to be forgotten” in Europe (General Data Protection Regulation (GDPR), Article 17, [9]) requires organizations to delete personal user data, including interaction data, upon request: “The data subject shall have the right to […] the erasure of personal data […] where the data subject withdraws consent.” Research on “machine unlearning” [4, 11, 16, 30, 35, 37, 38] argues that it is insufficient to delete personal data from primary data stores; machine learning models that have been trained on the stored data also fall under the regulation. Outside Europe, similar regulations are being adopted [1, 34].

The need for timely machine unlearning. GDPR does not specify how soon data must be erased after a deletion request [31], yet it states the “obligation to erase personal data without undue delay” [9] using “appropriate and effective measures” [10]. Currently, data erasure seems to be a rather tedious and lengthy process in practice; e.g., data erasure from active systems in the cloud can take up to two months [31] Not being able to enforce this right in a timely manner can have dramatic consequences in practice [36]. Therefore, we need to design search and recommendation approaches with timely “unlearning” capabilities, to empower users to quickly delete their interaction data and adjust their predictions on demand. For such unlearning methods to be effective in practice, they have to be (i) fast, to allow users to interactively delete their data from existing models, and (ii) exact, (e.g., no approximate updates, no hyperparameters to tune, no restriction on the maximum number of updates), to be easy to integrate into existing models. Unfortunately, machine unlearning is a hard problem, and existing general solutions either require partial iterative retraining of the underlying model [13, 16, 38] (and are therefore not fast), and/or can only conduct approximate updates for a small number of data points [13, 16, 20] (and are therefore not exact). Recent approaches to unlearning in recommendation such as RecEraser [5] inherit these limitations and conduct partial retraining, which can take several hours even for small datasets with 10 million interactions.

Fast and exact unlearning for neighborhood-based recommendation. Fast and exact unlearning is still possible under certain algorithmic and data-specific conditions. One such condition are $k$-nearest neighbor (kNN) models trained on sparse data. Such models are highly relevant for recommender systems, which work with extremely sparse interaction data. User-centric kNN models have a long tradition in this area, ranging from classical user-based collaborative filtering [28] to recent state-of-the-art algorithms in session-based [23, 24], session-aware [22], next-basket (NBR) [7, 14], and within-basket recommendation [3]. Moreover, compared to neural approaches to recommendation, kNN models are more transparent and explainable [24], cheap to scale to industry workloads [19] and often require an order of magnitude less training time [18].

Our contributions. We formalize the unlearning problem for kNN models in Section 2. Next, we detail caboose, an in-memory index of the top-$k$ most similar users in a sparse interaction dataset, which
enables unlearning of interactions in a fast and exact manner (Section 3). We achieve this by exploiting sparsity, choosing efficient data structures and parallelizing updates, while building upon existing work on fast indexing [32]. Our experimental evaluation in Section 4 shows that Caboose provides competitive index building times, makes sub-second unlearning possible (even for a large index built from 1 million users and 256 million interactions), and, when integrated into state-of-the-art NBR models, allows users to effectively adjust their predictions to remove sensitive items. Furthermore, we share an implementation of Caboose in Rust under an open license at https://github.com/amsterdata/caboose.

2 PROBLEM STATEMENT

We focus on user-centric kNN models for recommendation [3, 7, 14, 28]. These approaches typically model the interactions of $n$ users with $m$ items via a sparse matrix $U \in \mathbb{R}^{n \times m}$ and compute the top-$k$ similar users per user from this matrix, based on a similarity function between rows, which we denote $\Phi$. At training time, such user-centric kNN models build an index of the top-$k$ similar users per user. We denote this index with $\mathcal{T}_U$ and assume that it is built via a procedure $\text{build\_index}(U, \Phi, k)$. An entry $\mathcal{T}_U(i) = \{(u_{i,g}, \phi_{i,g}), \ldots, (u_{i,k}, \phi_{i,k})\}$ of this index for a user $u_i$ contains the $k$ most similar users $u_{i,g}, \ldots, u_{i,k}$ with their corresponding similarities $\phi_{i,g}, \ldots, \phi_{i,k}$. At inference time, such a top-$k$ index provides $O(1)$ access to the neighbors of a particular user.

In order to enable unlearning for user-centric kNN models, we have to answer the following research question: Given an existing index $\mathcal{T}_U$ and an interaction $u_{i,g}$ of a user $i$ with an item $g$ to unlearn, how can we efficiently compute $\mathcal{T}_U(U-u_{i,g})$?

This deletion affects more than just the entry $\mathcal{T}_U(i)$, as the user $u_i$ could be contained in the top-$k$ entries of many other users! We could obviously simply re-compute the top-$k$ index from scratch as $\mathcal{T}_U(U-u_{i,g}) = \text{build\_index}(U-u_{i,g}, \Phi, k)$. However, this is expensive, as recomputing the index from scratch can take many hours for large datasets, and wastes a lot of computation, as $\mathcal{T}_U$ and $\mathcal{T}_U(U-u_{i,g})$ are likely to only differ in a relatively small number of entries. Furthermore, this problem is not solved by existing vector indexes such as FAISS [27], Pinecone [26] or Hnswlib [8], which support deletions, but are designed for approximate search on dense vectors with a comparatively small number of dimensions only.

3 PROPOSED APPROACH

To tackle our research question, we design Caboose, an in-memory index of the top-$k$ most similar users in a sparse interaction dataset, which enables unlearning interactions in a fast and exact manner. Caboose provides an algorithm forget to unlearn an interaction $u_{i,g}$ from an existing index $\mathcal{T}_U$, which produces the same result $\mathcal{T}_U(U-u_{i,g})$ as re-computing the index from scratch without the interaction to unlearn:

$\mathcal{T}_U(U-u_{i,g}) = \text{forget}(\mathcal{T}_U, u_{i,g}, \Phi, k) = \text{build\_index}(U-u_{i,g}, \Phi, k)$.

At an abstract level, our FORGET algorithm proceeds in four stages to derive the updated index $\mathcal{T}_U(U-u_{i,g})$ from an existing index $\mathcal{T}_U$:

- **Stage 1**: Update $U$ to $U-u_{i,g}$, update $U^\top$ to $(U-u_{i,g})^\top$ and the precomputed norm of $u_i$ to reflect the deletion of $u_{i,g}$.
- **Stage 2**: Recompute the dot products $(u_i-u_{i,g})(U-u_{i,g})^\top$ involving $u_i$ to obtain its updated similarities and identify the set of affected top-$k$ entries $R$.
- **Stage 3**: Inspect and (where possible) directly update the top-$k$ entry $\mathcal{T}_U(r)$ for each row $r \in R$ affected by the deletion of $u_{i,g}$.
- **Stage 4**: Re-compute the top-$k$ entries from scratch for rows in $R$, which cannot be directly updated.

For this algorithm to be efficient in practice, we design it with the following characteristics in mind: (i) We exploit sparsity wherever possible to only work with the top-$k$ entries directly affected by the unlearning operation. Hence, we restrict our approach to similarity functions which are zero if the dot product between two rows is zero, and which can be computed from the dot product and norms of vectors. As a result, we can ignore pairs of users without a shared item interaction. This class involves many commonly used similarity measures, e.g., cosine similarity, computed as $\phi_{ij} = u_i^\top u_j/\|u_i\|\|u_j\|$ or Jaccard similarity. (ii) We parallelize the individual stages of our algorithm, and (iii) choose appropriate data structures that enable efficient low-level operations (e.g., compressed representations for sparse matrices and binary heaps for top-$k$ lists).

Index layout. The memory layout of Caboose is shown in Fig. 1. It enables efficient row-wise and column-wise access to $U$, by holding $U$ and $U^\top$ in compressed sparse row form (CSR). CSR represents a matrix with three arrays: the array data contains its non-zero values, and the array indices holds the corresponding column indices, and both are accessed through the array indptr which denotes the range of columns belonging to each row. Additionally, Caboose contains an array norms for the norm of each row (e.g., the $L^2$-norm for cosine similarity). The top-$k$ entries $\mathcal{T}_U$ are represented by the $n$-dimensional array topk, where each entry topk[i] represents $\mathcal{T}_U(i)$, and contains a binary heap of length $k$, which stores tuples of row identifiers and the corresponding similarity scores. The tuple with the $k$-largest similarity score is at the heap root, which enables us to check in $O(1)$ time if a new similarity tuple is among the $k$-largest tuples, and insert such a tuple with complexity $O(\log k)$. Each entry additionally contains a sorted array sorted_keys of length $k$ with the row identifiers from the heap to enable $O(\log k)$ membership tests for keys via binary search.

This design is inspired by [32] and we build our index accordingly: We precompute the norms for all rows, execute a sparse vector matrix multiplication $u_i U^\top$ for each row $u_i$ in parallel to obtain its dot products with other rows, calculate the final similarity based on the precomputed norms and extract the top-$k$ similar rows afterwards.

Unlearning algorithm. Next, we detail in Algorithm 1 how to efficiently conduct the four stages for unlearning an interaction $u_{i,g}$ of user $i$ with item $g$ from an existing index $\mathcal{T}_U$.

![Figure 1: Index layout in memory.](image)
Algorithm 1 Unlearning an interaction \(u_{ig}\) from the index \(\mathcal{T}_U\).

1: function forget_row(row i, column g)
2: \[
\text{norms}[i] \leftarrow \text{update_norm(norms[i], u_{ig})}
\] // Stage 1 - Updating \(U, U^T\) and the precomputed norms
3: set entry corresponding to \((i, g)\) in data to 0
4: set entry corresponding to \((g, i)\) in data to 0
5: \(A \leftarrow 0\) // Stage 2 - Parallel computation of \((u_i - u_{ig})(U - u_{ig})^T\)
6: \(\text{parfor } c \in [\text{indptr}[i], \ldots, \text{indptr}[i + 1])\) in partitions of size \(m/\text{cores} \text{do}\)
7: \(\text{a} \leftarrow \text{accumulator of capacity } n\)
8: for \(j \in c\) do \(\text{sparse_mult}(a, j)\)
9: \(A \leftarrow A \cup a\)
10: end parfor
11: \((R, \text{tnew}) \leftarrow \text{merge_and_collect}(i, k, \text{norms}, A)\)
12: \(\text{topk}[i] \leftarrow \text{tnew}\)
13: \(F \leftarrow 0\) // Stage 3 - Parallel update of affected top-k entries
14: \(\text{parfor } (r, \phi_r) \in R \text{ do}\)
15: if \(\text{find_with_binary_search}(i, \text{topk}[r].\text{sorted_keys})\) then
16: if \(\phi_r > \text{root of topk}[r].\text{heap}\) then \(\text{update_root(topk}[r], i, \phi_r)\)
17: else
18: if \(\phi_r \neq 0\) then
19: if \(\phi_r < \text{root of topk}[r].\text{heap}\) then \(F \leftarrow F \cup r\)
20: else \(\text{update_topk(topk}[r], (i, \phi_r))\)
21: end if
22: \(\text{if topk}[r] < k \text{ then remove_from_topk(topk}[r], i)\)
23: else \(F \leftarrow F \cup r\)
24: end if
25: \(\text{parfor } c \in F \text{ in partitions of size } 1024 \text{ do}\)
26: \(a \leftarrow \text{accumulator of capacity } n\)
27: for \(r \in c\) do
28: for \(j \in [\text{indptr}[r], \ldots, \text{indptr}[r + 1])\) \(\text{do}\) \(\text{sparse_mult}(a, j)\)
29: \(\text{topk}[r] \leftarrow \text{topk_and_clear}(a, r, k, \text{norms})\)
30: end parfor
31: \(\text{function sparse_mult(accumulator a, pointer j)}\)
32: \(\text{for } l \in [\text{indptr}_1[\text{indices}[j]], \ldots, \text{indptr}_1[\text{indices}[j + 1])\) \(\text{do}\)
33: \(\text{accumulate}(a, \text{indices}_1[l], \text{data} t[l] \cdot \text{data} [l])\)

Stage 1 - Updating \(U, U^T\) and the precomputed norms. Lines 2–4 in Algorithm 1 update the stored norm of \(u_i\) according to the norm required for the similarity (e.g., to \(\sqrt{\text{norms}[i]^2 - w_{ig}^2}\) for cosine similarity) and set the entries corresponding to \(u_{ig}\) in the CSR representations of \(U\) and \(U^T\) to 0.

Stage 2 - Parallel dot product updates \((u_i - u_{ig})(U - u_{ig})^T\). Next, we recompute the dot products for the updated row \(u_i\) by computing \((u_i - u_{ig})(U - u_{ig})^T\) via sparse parallel vector matrix multiplication in Lines 5–10. From the result, we obtain the updated top-k similarities \(\text{tnew}(\text{representing } \mathcal{T}_U(\cdot, u_{ig}))\), updated similarities and affected rows \(R\) (which have a non-zero dot product with \(u_{ig}\)) (Line 11).

Stage 3 - Parallel updates of affected top-k entries. Next, we inspect and potentially change each entry \(\mathcal{T}_U(r)\) for an affected row \(u_r \in R\) with an updated similarity \(\phi_r\) between \(u_i\) and \(u_r\) (Lines 13–24). Note that we need to distinguish several cases here:

- Not-in-top-k: In Line 15 we first test whether the row \(u_i\) is already part of the top-k entry \(\mathcal{T}_U(r)\) of \(u_i\) via binary search on \(\text{topk}[r].\text{sorted_keys}\). If \(u_i\) is not contained, we check whether the unlearning operation changed \(u_i\) to be part of the top-k entry of \(u_i\) by comparing the new similarity \(\phi_r\) to the heap root of \(\text{topk}[r]\) (the k-largest similarity to \(u_i\)). If this holds, we update the corresponding \(\mathcal{T}_U\) entry via \(\text{update_root}\) in Line 16.

- Already-in-top-k: If \(u_i\) is already in the top-k of \(u_r\) and the updated similarity \(\phi_r\) is non-zero, then we have to distinguish two cases: \((i)\) if \(\phi_r\) is not smaller than the current heap root, we can simply recreate the corresponding heap to reflect the changed similarity (Line 20). Otherwise, \((ii)\) the \(k\) th element (which is not contained in the index) might actually be larger than the updated one for \(u_i\), therefore we need to recompute the top-k entry for \(u_r\) from scratch. We add \(r\) to the set \(F\) to schedule this recomputation later (Line 19).

- To-be-removed-from-top-k: The trickiest case is when \(u_i\) has been in the top-k of \(u_r\) already, but the unlearning operation results in a zero similarity \(\phi_r\) (Line 21). This happens if \(g\) is the last shared non-zero column between \(u_i\) and \(u_r\). We distinguish two cases here: \((i)\) if there are less than \(k\) similar rows for \(u_r\) in the data anyway, we can just delete \(u_i\) from the top-k entry of \(u_r\) via \(\text{remove_from_topk}\) (Line 22). However, \((ii)\) if the top-k entry of \(u_r\) has \(k\) elements, then we would need to remove \(u_i\) and replace it with the \((k+1)\)-st most similar row (which is not stored in the current index). As a consequence, we again need to recompute the top-k entry for \(u_r\) from scratch, and add \(r\) to the set \(F\) to schedule this recomputation later (Line 23).

Stage 4 - Parallel recomputation of non-updatable top-k entries. We finally recompute the non-updatable top-k entries contained in the set \(F\) from scratch via parallel sparse vector matrix multiplications in Lines 25–30. We empirically find this to happen rarely.

Due to lack of space, we refer to our shared code repository for details on the accumulators and update functions.

### 4 EXPERIMENTAL EVALUATION

We implement caboose in Rust and evaluate it on five sparse interaction datasets listed in Table 1. We use cosine similarity for the index, and if not reported otherwise, run experiments on a machine with a four-core Intel i7-8569U CPU @2.80GHz, 16GB of RAM and MacOS 12.6. Our experiment code is available at https://github.com/amsterdata/caboose.

#### Experiment 1: Index building time. In our first experiment, we showcase that caboose can be built in a time that is competitive with existing approaches. We compute the top-k similar users for all datasets with \(k = 50\) using our index and two baselines: similaripy [32] from the RecSys Challenge 2018, and unsupervised nearest neighbors from sklearn [25]. We repeat each run seven times and report the mean time to build the index. Note that we run the experiments for spotify and yahoo on a larger machine with 64 AMD EPYC 7H12 2.6 GHz cores and AlmaLinux 8.6, as the experiment would take several days otherwise.

#### Results and discussion. Figure 2 plots the mean runtimes in milliseconds on a logarithmic scale. We had to cancel the experiments for sklearn on the large datasets, as this implementation did not manage to make proper use of all cores of the system. We find that caboose outperforms sklearn in all cases by a factor of up to 2.26 and similaripy in three out of five datasets by a factor of up to 2.27. We attribute the runtime differences with similaripy...
We build indexes with $k$ we randomly choose 500 interactions to forget and measure the unlearning time [ms]. We assume that a user does not want to see personalized recommendations. We evaluate a simple strategy in an e-commerce setting: We assume that a user has sensitive items in their predictions, and we cannot include such products from a certain sensitive product category, even though they interacted with such products in the past. If the user is exposed to recommendations with such sensitive items, they ask the system to unlearn their past interactions with products from this sensitive category. We use the instacart30k [15] dataset for grocery shopping as a basis for the experiment. We define three types of sensitive product categories: (i) baby items motivated by a report on a recent traumatizing case [36], (ii) meat-related categories to mimic a person changing to a vegetarian diet, and (iii) alcohol-related categories to mimic a person suffering from alcohol addiction.

For each sensitive category, we sample the baskets of 1,000 users who bought at least one item from this category, determine the other non-sensitive categories in their baskets, and sample 1,000 additional users who bought items from these categories but not from the sensitive category. We train the models on this dataset of 2,000 users (with default hyperparameters), and inspect the top-10 predicted items for each user with sensitive purchases. If such a user has sensitive items in their predictions, we make the model forget all the user’s historical interactions with items from the sensitive category, recompute the predictions for the user and check if any sensitive items remain in the top-10 predictions. We repeat this experiment with seven random seeds for the three models and three categories, and report the mean number of affected users (who initially had sensitive items in their recommendations) and the fraction of such users where the simple unlearning strategy successfully removed the sensitive items from their recommendations.

Table 2: Impact of unlearning sensitive items from user histories to remove sensitive items from their predictions.

<table>
<thead>
<tr>
<th>Sensitive category</th>
<th>Users affected</th>
<th>Removal success</th>
</tr>
</thead>
<tbody>
<tr>
<td>baby items</td>
<td>22.4%</td>
<td>99.80%</td>
</tr>
<tr>
<td>meat</td>
<td>36.3%</td>
<td>99.79%</td>
</tr>
<tr>
<td>alcohol</td>
<td>33.1%</td>
<td>99.62%</td>
</tr>
</tbody>
</table>

Results and discussion. We detail the results of this experiment in Table 2. Across all models and categories, a large fraction (between 99.5% and 100%) of users who interacted with sensitive items also get exposed to sensitive items in their top-10 predictions. The simple removal strategy of forgetting past interactions with sensitive items is very effective, and removes almost all sensitive items from the predictions. It fails in a small number of cases (mostly for alcohol), which we attribute to the strong cooccurrence between an alcoholic beverage and non-alcoholic items like certain dishes in the data; here the user would have to forget these interactions as well.

5 CONCLUSION

We formalized the problem of unlearning for KNN models on sparse data, and detailed how to unlearn interactions in a fast and exact manner from a custom in-memory top-$k$ index. For future work, we plan to integrate GraphBLAS [6] for accelerating our sparse operations. We aim to avoid the potential full recomputation of individual top-$k$ entries from a “budget” of more than the required $k$ most similar users computed at indexing time.

Acknowledgements. This work was supported by Ahold Delhaize. All content represents the opinion of the authors, which is not necessarily shared or endorsed by their respective employers and/or sponsors.
REFERENCES