Serenade - Low-Latency Session-Based Recommendation in e-Commerce at Scale

Barrie Kersbergen
bkersbergen@bol.com
bol.com

Olivier Sprangers
o.r.sprangers@uva.nl
AIRLab, University of Amsterdam

Sebastian Schelter
s.schelter@uva.nl
University of Amsterdam

ABSTRACT

Session-based recommendation predicts the next item with which a user will interact, given a sequence of her past interactions with other items. This machine learning problem targets a core scenario in e-commerce platforms, which aim to recommend interesting items to buy to users browsing the site. Session-based recommenders are difficult to scale due to their exponentially large input space of potential sessions. This impedes offline precomputation of the recommendations, and implies the necessity to maintain state during the online computation of next-item recommendations.

We propose *VMIS-kNN*, an adaptation of a state-of-the-art nearest neighbor approach to session-based recommendation, which leverages a prebuilt index to compute next-item recommendations with low latency in scenarios with hundreds of millions of clicks to search through. Based on this approach, we design and implement the scalable session-based recommender system *Serenade*, which is in production usage at *bol.com*, a large European e-commerce platform.

We evaluate the predictive performance of VMIS-kNN, and show that Serenade can answer a thousand recommendation requests per second with a 90th percentile latency of less than seven milliseconds in scenarios with millions of items to recommend. Furthermore, we present results from a three week long online A/B test with up to 600 requests per second for 6.5 million distinct items on more than 45 million user sessions from our e-commerce platform. To the best of our knowledge, we provide the first empirical evidence that the superior predictive performance of nearest neighbor approaches to session-based recommendation in offline evaluations translates to superior performance in a real world e-commerce setting.

ACM Reference Format:
Barrie Kersbergen, Olivier Sprangers, and Sebastian Schelter. 2022. Serenade - Low-Latency Session-Based Recommendation in e-Commerce at Scale. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3514221.3517901

SIGMOD '22, June 12–17, 2022, Philadelphia, PA, USA

ACM ISBN 978-1-4503-9249-5/22/06…$15.00
https://doi.org/10.1145/3514221.3517901

1 INTRODUCTION

Session-based recommendation targets a core scenario in e-commerce and online browsing. Given a sequence of interactions of a visitor with a selection of items, we want to recommend to the user the next item(s) of interest to interact with [27, 29, 30, 37]. This machine learning problem is crucial for e-commerce platforms [24].

Challenges in scaling session-based recommendation. Scaling session-based recommender systems is a difficult undertaking, because the input space (sequences of item interactions) for the recommender system is exponentially large (of size $|\mathbf{I}|^n$ for all possible sessions of length n from a set of items I), which renders it impractical to precompute recommendations offline and serve them from a data store. This is in stark contrast to classical collaborative-filtering based recommendations [25, 39], which are relatively static as they rely on long-term user behavior [40]. Instead, session-based recommenders have to maintain state in order to react to online changes in the evolving user sessions, and compute next item recommendations with low latency [9, 24] in real-time.

Recent research indicates that nearest neighbor methods provide state-of-the-art performance for session-based recommendation, and even outperform complex neural network-based approaches in offline evaluations [24, 30]. It is however unclear whether this superior offline performance also translates to increased user engagement in real-world recommender systems. Furthermore, it is unclear whether the academic nearest neighbor approaches scale to industrial use cases, where they have to efficiently search through hundreds of millions of historical clicks while adhering to strict service-level-agreements for response latency. This scalability challenge is further complicated by the fact that the applied session similarity functions do not constitute a metric space (e.g., due to lack of symmetry), which renders common approximate nearest neighbor search techniques inapplicable.

VMIS-kNN. In order to tackle the scalability challenge, we present *Vector-Multiplication-Indexed-Session-kNN* (VMIS-kNN) in Section 3, an adaption of the state-of-the-art session-based recommendation algorithm VS-kNN [30]. VMIS-kNN leverages a prebuilt index to compute next-item recommendations in milliseconds for scenarios with hundreds of millions of clicks in historical sessions to search through. Our approach can be viewed as the joint execution of a join between evolving and historical sessions on matching items and two aggregations to compute the similarities. During this joint execution, we minimise intermediate results, control the memory usage and prune the search space with early stopping. As a consequence, VMIS-kNN drastically outperforms VS-kNN in terms of latency and scalability (Section 5.2.1), while still providing the desired prediction quality advantages over neural network-based approaches (Section 5.1.1).

Serenade. Finally, we present the design and implementation of our scalable session-based recommender system *Serenade*, which employs VMIS-kNN, and can serve a thousand recommendation requests per second with a 90th percentile latency of less than seven milliseconds in scenarios with millions of items to recommend. Our system runs in the Google Cloud-based infrastructure of *bol.com*, a large European e-commerce platform, and is in production usage. We discuss design decisions of Serenade, such as stateful recommendation servers, which colocate the evolving user sessions together with update and recommendation requests (Section 4.1). Additionally, we describe implementation and deployment details (Section 4.2), as well as insights into the remarkably low operational costs for our system (Section 7).

Offline and online evaluation. We conduct an extensive evaluation to validate the predictive performance and low latency of VMIS-kNN in Section 5.1. For the Serenade system, we present results from a load test with more than 1,000 requests per second, and the outcome of a three week long online A/B test of our system on the live e-commerce platform in Section 5.2. Our system is available under an open license at https://github.com/bolcom/serenade.

In summary, we provide the following contributions.

- We present VMIS-kNN, an index-based variant of a state-of-the-art nearest neighbor algorithm to session-based recommendation, which scales to use cases with hundreds of millions of clicks to search through (Section 3).
- We discuss design decisions and implementation details of our production recommender system Serenade, which applies stateful session-based recommendation with VMIS-kNN, and can handle more than 1,000 requests per second with a response latency of less than seven milliseconds in the 90th percentile (Section 4).
- To the best of our knowledge, we provide the first empirical evidence that the superior predictive performance of VMIS-kNN/VS-kNN from offline evaluations translates to superior performance in a real world e-commerce setting; we find Serenade to drastically increase a business-specific engagement metric by several percent, compared to our legacy system (Section 5.2.3).

2 BACKGROUND

We introduce session-based recommendation and the Vector-Session-kNN method. Given an evolving session (a sequence of interactions with a set of items $\mathbf{I}$) at time t, the goal of session-based recommendation is to accurately predict the next item that the user will interact with at time $t+1$.

Vector-Session-kNN. Vector-Session kNN (VS-kNN) [30] is a state-of-the-art nearest neighbor based approach to session-based recommendation, which outperforms current deep learning approaches for this task. In VS-kNN, we have a set of historical sessions $\mathbf{H} \in \{0,1\}^{|\mathbf{I}|}$ represented as binary vectors in item space, and an evolving user session $\mathbf{s}^{(t)} \in \{0,1\}^{|\mathbf{I}|}$ at time t, as well as a function $\omega(\mathbf{s})$ which replaces the non-zero entries of $\mathbf{s}$ with integers denoting the insertion order of the items in $\mathbf{s}^{(t)}$. Algorithm 1 describes how VS-kNN computes its recommendations for an evolving session $\mathbf{s}^{(t)}$. First a recency-based sample $\overline{\mathbf{H}}_s$ of size m is taken from all historical sessions $\mathbf{H}_s$ that share at least one item with

Algorithm 1 Vector-Session-kNN.

1: **function** VS-KNN($\mathbf{s}^{(t)}, \mathbf{H}, \pi, \lambda, m, k$)
2: *Input:* Evolving session $\mathbf{s}^{(t)}$, set of historical sessions $\mathbf{H}$, decay function π,
3: match weight function λ, sample size m, number of neighbors k.
4: *Output:* Scored list of recommended next items $\mathbf{d}$.
5: $\mathbf{H}_s \leftarrow$ historical sessions that share at least one item with $\mathbf{s}$
6: $\overline{\mathbf{H}}_s \leftarrow$ recency-based sample of size m from $\mathbf{H}_s$
7: $\mathbf{N}_s \leftarrow k$ closest sessions $\mathbf{h} \in \overline{\mathbf{H}}_s$ according to similarity $\pi(\omega(\mathbf{s}^{(t)}))^\top \mathbf{h}$
8: **for** each item i occuring in the sessions $\mathbf{N}_s$ **do**
9: $d_i \leftarrow \sum_{\mathbf{n} \in \mathbf{N}_s} \mathbb{1}_n(i) \cdot \frac{1}{|\mathbf{s}^{(t)}|} \cdot \lambda(\max(\omega(\mathbf{s}^{(t)}) \odot \mathbf{n})) \cdot \pi(\omega(\mathbf{s}^{(t)}))^\top \mathbf{n} \cdot (1 + \log \frac{|H|}{h_i})$
return item scores $\mathbf{d}$

the evolving session (Lines 5 & 6). Next, we compute the k closest sessions $\mathbf{N}_s$ from $\overline{\mathbf{H}}_s$ according to the similarity $\pi(\omega(\mathbf{s}^{(t)}))^\top \mathbf{h}$ (Line 7), which applies an element-wise decay function π to the entries denoting the insertion order in the evolving session. All items occurring in these neighboring sessions are finally scored (Lines 8 & 9) by summing their similarities (the previously computed decayed dot product) weighted by a non-linear function λ applied to the position $\max(\omega(\mathbf{s}^{(t)}) \odot \mathbf{n})$ of the most recent shared item between the evolving session $\mathbf{s}^{(t)}$ and the neighbor session $\mathbf{n}$. The session similarity contribution is additionally weighted by a factor of one over the session length, and by a factor of one plus the "inverse document frequency" $\log \frac{|H|}{h_i}$ of the item, where h_i denotes the number of historical sessions containing item i (a common technique from information retrieval to de-emphasise highly frequent items). Note that the indicator function $\mathbb{1}_n(i)$ is one if item i occurs in the historical session $\mathbf{n}$ and zero otherwise.

Toy example. We provide a toy example for the session similarity and match weighting computation executed by VS-kNN. Assume that we have an evolving session $\mathbf{s}^{(t)} = [0\ 1\ 1\ 0\ 1]$ representing interactions with the three items `[1,2,4]` and a historical session $\mathbf{h} = [0\ 0\ 1\ 0\ 1]$ representing interaction with the items `[2,4]`. The function ω gives us the chronological insertion order for the evolving session, e.g., $\omega(\mathbf{s}^{(t)}) = [0\ 1\ 2\ 0\ 3]$ of the items in $\mathbf{s}^{(t)}$, starting from the first item (item 1 with insertion time 1) to the most recent item (item 4 with insertion time 3). The insertion order is used to weight matches between the items of the evolving session and the historical session, and the weights are determined by the decay function π, which is a hyperparameter of VS-kNN. A common choice for π is to divide the insertion time by the session length, e.g., $\pi(\omega(\mathbf{s}^{(t)})_i) = \omega(\mathbf{s}^{(t)})_i \,/\, ||\mathbf{s}^{(t)}||_1$. The similarity is finally determined by computing the decayed dot product $\pi(\omega(\mathbf{s}^{(t)}))^\top \mathbf{h}$ between the evolving session $\mathbf{s}^{(t)}$ and historical session $\mathbf{h}$ as the sum of the decayed weights for the intersection of the sessions (the shared items), e.g., $\pi(\omega(\mathbf{s}^{(t)}))^\top \mathbf{h} = [0\ \frac{1}{3}\ \frac{2}{3}\ 0\ \frac{3}{3}]^\top [0\ 0\ 1\ 0\ 1] = \frac{2}{3} + \frac{3}{3} = \frac{5}{3}$.

After finishing the session similarity computation, VS-kNN computes item scores from the similarities (Lines 8 & 9). The score for an item is the weighted sum of similarities with $\mathbf{s}^{(t)}$ from the k closest historical sessions $\mathbf{n} \in \mathbf{N}_s$ in which the item occurs. The weights for this sum are computed by the matching function λ, which is applied to the insertion time $\max(\omega(\mathbf{s}^{(t)}) \odot \mathbf{n})$ of the most recent shared item between $\mathbf{s}^{(t)}$ and $\mathbf{n}$. The default

choice for λ in VS-kNN is $1 - (0.1 \cdot (\max(\omega(\mathbf{s}^{(t)}) \odot \mathbf{n})))$ for insertion times less than 10 and zero otherwise. For our toy example, the contribution of the matching function for $\mathbf{h}$ looks as follows: $\lambda(\max(\omega(\mathbf{s}^{(t)}) \odot \mathbf{h})) = \lambda(\max(\,[0\ 1\ 2\ 0\ 3]\ \odot\ [0\ 0\ 1\ 0\ 1]\,)) = \lambda(\max(\,[0\ 0\ 2\ 0\ 3]\,)) = \lambda(3) = 0.7$.

3 VECTOR-MULTIPLICATION-INDEXED-SESSION-KNN (VMIS-KNN)

In the following, we present our scalable, index-based adaption of VS-kNN, which we call *Vector-Multiplication-Indexed-Session-kNN* (VMIS-kNN).

VMIS-kNN operates on an index structure $(\mathbf{M}, \mathbf{t})$, which we build from a large dataset of historical sessions. We create a hash index $\mathbf{M}$ from an item i to an array $\mathbf{m}_i$ of the m most recent historical sessions in which the item occurs. Note that m is a hyperparameter of VMIS-kNN, which denotes the size of the recency-based sample from which session similarity candidates are taken. Each array $\mathbf{m}_i$ of session identifiers for an item i is stored in descending timestamp order of the sessions (i.e., the most recent historical session h that contained the item i is the first entry in the vector $\mathbf{m}_i$). The key benefit of this data structure is to allow us amortised constant-time access to the m most recent sessions containing an item.

Furthermore, we maintain an array $\mathbf{t}$ where an entry t_h denotes the integer timestamp for a historical session h. This again provides constant time random access during the online computation of the session similarity score across all the items in an evolving session, as we use consecutive integer identifiers for historical sessions. Algorithm 2 describes the individual steps and data structures that VMIS-kNN leverages for efficient session-based recommendation based on our index data structure.

Index-based session similarity computation. At the heart of VMIS-kNN is the efficient computation of the neighbor sessions $\mathbf{N}_s$ for an evolving session $\mathbf{s}^{(t)}$ using our previously introduced index structure $(\mathbf{M}, \mathbf{t})$ in the function neighbor_sessions_from_index in Line 8.

We first initialize a set of temporary hashmaps and heaps (Line 11) which serve as buffers for intermediate results during the computation. Next, VMIS-kNN starts the *item intersection loop*, which iterates over the items in an evolving session $\mathbf{s}^{(t)}$ in reverse order (Line 12). Our approach processes an evolving session $\mathbf{s}^{(t)}$ in inverse insertion order, such that the most recent (and therefore most important) items of an evolving session are visited first. We then add the item identifier i to the temporary hashset $\mathbf{d}$, such that duplicate items in the evolving session can be skipped (Lines 13-14). Next, we look up the item in our inverted index $\mathbf{M}$ to obtain the vector $\mathbf{m}_i$ containing up to m historical session identifiers (Line 15). We then compute the decay score π_i based on the item's position in the evolving session (Line 16).

Now, we start a loop over each historical session j in $\mathbf{m}_i$ (Line 17). If we have already encountered this historical session for a different item, we add the current decay score π_i to the session score r_j (Line 18). However, if the historical session is not yet part of our temporary similarity score hashmap $\mathbf{r}$, we first obtain the timestamp t_j of the historical session (Line 20). If our temporary similarity score hashmap $\mathbf{r}$ contains less than m items, we insert the session identifier j and session similarity score r_j as (key, value)-pair into $\mathbf{r}$, and we insert the session identifier j and session timestamp t_j as (key, value)-pair into a min-heap $\mathbf{b}_t$ (Lines 21-24). If our temporary similarity score hashmap $\mathbf{r}$ already contains m sessions, we need to investigate whether to remove the oldest session. Therefore, we first retrieve the oldest session and corresponding timestamp from the heap $\mathbf{b}_t$ (Line 26).

If the current historical session j is more recent than the oldest session, we need to remove the oldest session from our temporary similarity score hashmap $\mathbf{r}$ and heap $\mathbf{b}_t$, and update both with the values from the current historical session j (Lines 27-31). Finally, we extract the top-k scored sessions from the max-heap $\mathbf{N}_s$ in the *top-k similarity loop* and return them (Line 33).

VMIS-kNN computes the final item scores by using the precomputed session similarity r_n for a neighboring historical session $\mathbf{n}$. We however simplify the item scoring function from Line 9 of Algorithm 1 in two ways: (i) we remove the constant factor $1/|\mathbf{s}^{(t)}|$ applied to each similarity (which does not change the neighbor ranking); and (ii) we use a weight of $\log \frac{|H|}{h_i}$ instead of $(1 + \log \frac{|H|}{h_i})$ for the similarities, which gives us better results in offline evaluations on held-out data.

Algorithm 2 Vector-Multiplication-Indexed-Session-kNN.

```
1:  function VMIS-KNN(s^(t), (M, t), π, λ, m, k)
2:      Input: Evolving session s^(t), session similarity index (M, t), decay function π,
3:          sample size m, match weight function λ, number of neighbors k.
4:      Output: Scored list of recommended next items d.
5:      (N_s, r) ← neighbor_sessions_from_index(s^(t), (M, t), π, m, k)
6:      for each item i occuring in the sessions N_s do
7:          d_i ← Σ_{n∈N_s} 𝟙_n(i) · λ(max(ω(s^(t)) ⊙ n)) · r_n · log |H|/h_i
        return item scores d

8:  function NEIGHBOR_SESSIONS_FROM_INDEX(s^(t), (M, t), π, m, k)
9:      initialize hashmap r for temporary similarity scores, min-heap b_t of cap-
10:     acity m for the most recent similar historical sessions, hashset d for already
11:     processed items, max-heap N_s of capacity k for closest sessions
12:     for item i ∈ s^(t) in reverse insertion order do          ▷ Item intersection loop
13:         if i ∉ d then
14:             insert i into d
15:             m_i ← most recent sessions for item i from inverted index M
16:             π_i ← decay weight π(ω(s^(t)))_i of item i in session s^(t)
17:             for session j ∈ m_i do
18:                 if j ∈ keys(r) then r_j ← r_j + π_i
19:                 else
20:                     t_j ← timestamp of session j fetched from index t
21:                     if |r| < m then
22:                         r_j ← π_i
23:                         insert (j, r_j) into r
24:                         insert (j, t_j) into b_t
25:                     else
26:                         (l, t_l) ← current heap root of b_t
27:                         if t_j > t_l then
28:                             r_j ← π_i
29:                             remove (l, r_l) from r
30:                             insert (j, r_j) into r
31:                             update heap root of b_t with (j, t_j)
32:                         else break

33:     for (j, r_j) ∈ r do                                      ▷ Top-k similarity loop
34:         if |N_s| < k then insert (j, r_j) into N_s
35:         else
36:             (n, r_n) ← current heap root of N_s
37:             if r_j > r_n then update heap root of N_s with (j, r_j)
38:             else if r_j = r_n and t_j > t_n then update heap root of N_s with (j, r_j)
39:     return N_s
```

A particular advantage of VMIS-kNN is its support for early stopping, which allows us to skip certain historical sessions during the similarity computation: we can immediately break the session for-loop if our current historical session j is older than the eldest session l in our heap $\mathbf{b}_t$ as $\mathbf{m}_i$ is already sorted in descending timestamp order, and will not contain more recent sessions in later positions (Line 32).

Time complexity. The time complexity of the online computation of our similarity score is dominated by the linear time required to execute the three for-loops (Lines 12, 17 and 33) and the logarithmic time required to modify the heaps $\mathbf{b}_t$ (Lines 24, 31) and $\mathbf{N}_s$ (Lines 34, 37, 38), yielding a theoretical time complexity of $O(|\mathbf{s}^{(t)}| \cdot m \cdot \log_2 m + m \cdot \log_2 k) = O(|\mathbf{s}^{(t)}| \cdot m \cdot \log_2 m)$ as $k \leq m$. Thus, the time complexity only depends on: (i) the number of items in the evolving session $\mathbf{s}^{(t)}$, which we cap at a maximum value, and (ii) the number m denoting how many recent historical sessions to consider. Hence, the time complexity of our implementation is (theoretically) independent of the number of historical sessions $|\mathbf{H}|$ and the number of unique items $|\mathbf{I}|$ in our dataset. As a micro-optimisation, we leverage octonary heaps [2] instead of binary heaps, which have more children per node, and therefore provide better performance for workloads like ours with frequent insertions.

Space complexity. The space complexity of the index for VMIS-kNN is dominated by the storage cost $O(|\mathbf{I}| \cdot m)$ for the hashmap holding the inverted index $\mathbf{M}$, which maps unique item indices to the m most recent historical sessions containing the item.

From a classical query processing perspective, VMIS-kNN conducts two aggregations (identifying the m most recent sessions with an item match, and computing their similarities) on the result of a join between the items of the evolving session $\mathbf{s}^{(t)}$ and the historical sessions $\mathbf{H}$. The efficiency of VMIS-kNN derives from the fact that we jointly execute the join and subsequent aggregations in Algorithm 2, while only maintaining intermediate results of a size proportional to the final outputs (instead of first materialising the potentially large complete join result before running the aggregations).

4 SERENADE

We present the design and implementation of our scalable recommender system *Serenade*, which leverages VMIS-kNN (Section 3) and provides recommendations on the product detail pages of bol.com.

4.1 Design Considerations

At the core of the design of our production system are two questions: (i) How to maintain the session similarity index over time; and (ii) How to efficiently serve next-item recommendations with low latency?

Index maintenance. We execute the index computation in an offline manner once per day with a data-parallel implementation of the relational operations required for the index generation. This batch job is easy to schedule and scale; note that Serenade will thus only see sessions for new items on the platform with a delay of one day. This "cold-start" issue is no problem in practice however, because our e-commerce platform has a separate, specialised system for presenting new and trending items to users.

Low latency serving of next-item recommendations. The biggest challenge in our system is to serve session-based recommendations with a low latency for a catalog containing millions of items (our business constraint is to respond in 50 ms or less for at least 90% of all requests). As discussed in Section 1, we cannot precompute the recommendations due to the exponentially large input space of potential sessions, and we cannot apply approximate nearest neighor search techniques because our similarity function is not a metric. As a consequence, our recommendation servers have to be stateful, by maintaining copies of the evolving sessions, to be able to compute recommendations online on request. We decide to replicate our session index to all recommendation servers, and colocate the session storage with the update and recommendation requests, so that we only have to use machine-local reads and writes for maintaining sessions and computing recommendations. Note that similar techniques are often used to accelerate joins [16].

4.2 Implementation

The high-level architecture of Serenade (derived from our design decisions in Section 4.1) is illustrated in Figure 1. Serenade consists of two components: The offline component (shown in the left part of the figure) builds the session index from click data and is implemented as an Apache Spark pipeline. The online component (shown in the right part of the figure) computes and serves session-based recommendations with VMIS-kNN, and is implemented as a REST application. Note that Serenade builds upon existing Google Cloud infrastructure rented by bol.com.

Offline index generation. The index generation ❶ from historical click data is implemented as a parallel dataflow computation in Apache Spark using Spark MLLib pipeline steps [33] as abstraction, and executed in regular intervals (typically once per day) in Google Dataproc. It uses historical click data from the last 180 days of our platform (stored in Google BigQuery) as its input, which amounts to roughly 2.3 billion user-item interactions. The output of the Spark job is a compressed representation of our index, stored in the distributed filesystem in the Apache Avro format. The index data is later on ingested by Serenade's serving component, where it requires around 13 gigabytes of memory.

Online serving of next item recommendations. The serving component of Serenade is responsible for computing next item recommendations with VMIS-kNN in response to session updates. We implement this serving component in Rust, as a web application based on the Actix [1] framework. The shopping frontend contacts our Serenade servers whenever a user generates new item interactions in their session (e.g., by visiting a product detail page). The Serenade servers update the state of the evolving user session ❷, and respond to the shopping frontend with a list of 21 recommended next items for the user (the number of items required by the UI in the frontend) based on a VMIS-kNN prediction ❸. The VMIS-kNN predictions leverage the previously computed offline index. We additionally apply business rules to the recommendations to remove unavailable products and to filter for adult products.

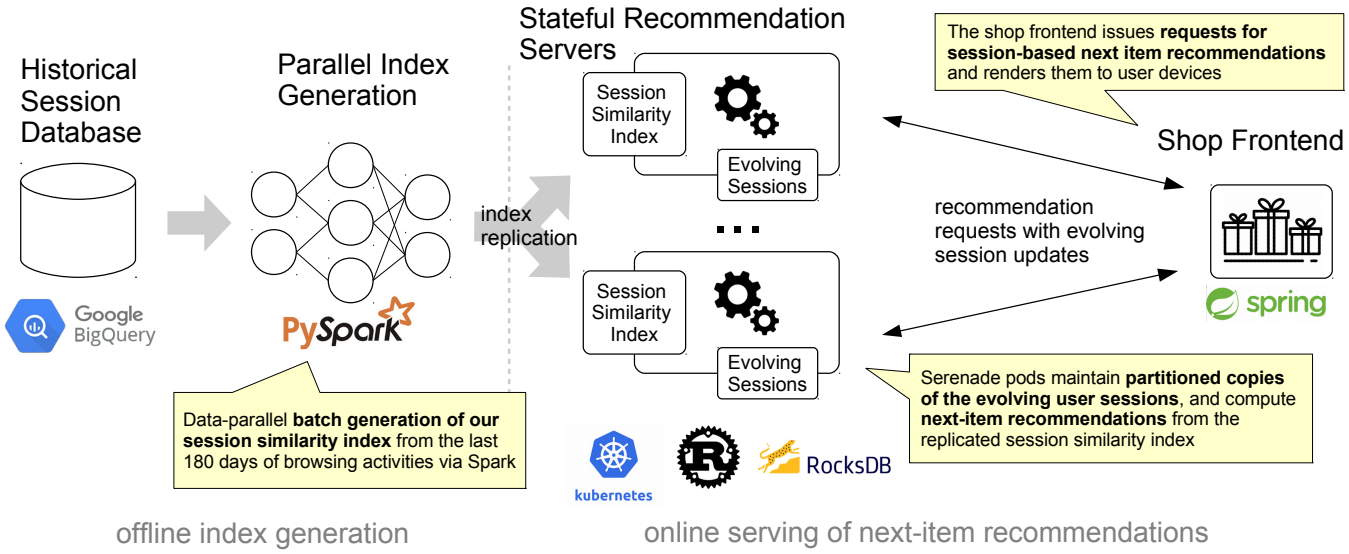


Figure 1: High level architecture of the Serenade recommendation system. The offline component (left) generates a session similarity index ❶ from several billion historical click events via a parallel Spark job in regular intervals. The online serving machines (right) maintain state about the evolving user sessions ❷, and leverage the session similarity index to compute next item recommendations with VMIS-kNN in response to recommendation requests from the shopping frontend ❸.

Colocation of evolving sessions and session updates. As discussed in Section 4.1, we need to colocate the evolving sessions with the recommendation requests and session updates to be able to compute up-to-date recommendations with low latency. We maintain the evolving sessions in a local key-value store (RocksDB [6]) directly on the serving machines, to avoid additional network reads and writes. For colocation, we have to partition both the evolving sessions and the recommendation requests (which also contain the session updates) over the serving machines, based on their session identifier. In order to guarantee that all the update/recommendation requests for a particular session are always handled by the same machine, we configure request routing via "sticky sessions" provided by Kubernetes' session affinity functionality [4]. The communication with RocksDB turns out to be extremely fast; in a microbenchmark with 10 million operations for our workload, we found the 99th percentile of the read latency to be 5 microseconds, and the 99th percentile of the write latency to be 18 microseconds. This colocation approach provides a big latency improvement over network reads and writes to a distributed key-value store like BigTable, where the response latency for lookups is already 15ms on the 99.5 percentile in our experience.

Discussion. Our colocation approach can be viewed as a trade-off between reducing the response latency and guaranteeing fault tolerance for the session data, as the session data could be temporarily lost in cases of machines failures or elastic scaling of the machine pool. However, this turns out to be no problem in practice for several reasons: (*i*) Our service proved to be very stable, we encountered no issues in a long A/B test running for several weeks (details will be described in Section 5.2.3), where no elastic scaling was required, as a small set of cheap machines with a low number of cores could reliably handle hundreds of request per second; (*ii*) The sessions are very short-lived anyways, we only leverage the most recent interactions for recommendations (which also have the highest impact on the session similarities), their loss would not have a drastic impact, as the recommender would quickly collect new interactions; (*iii*) The sessions are additionally tracked by other parts of our e-commerce platform for analytics. It is not the task of the recommendation system to store them permanently, on the contrary, we configure RocksDB to remove the data for a session after 30 minutes of inactivity.

Deployment. We deploy our recommendation servers via a Docker image managed by Kubernetes. The image is created by our continuous integration infrastructure, and we leverage a multi-stage build. In the first stage, we download all dependencies and compile our Rust application (which results in a large image with a size of several gigabytes); in the second stage, we reduce the size of this image by only retaining the compiled application and the runtime dependencies. The image for Serenade is then pushed into a Docker repository. The application is deployed to a Google Kubernetes Engine cluster, alongside with load balancing pods (`istio sidecars` [3]) which provide us with the session affinity routing required for colocating the evolving user sessions and recommendation requests on our machines.

Depersonalisation. We are required to provide non-personalised recommendations for users who do not give consent to leverage their session history for personalisation. This is comparatively easy to implement with VMIS-kNN: we create a non-personalised variant which only leverages the currently displayed item on the product detail page for recommendation. This depersonalisation can be applied in real-time (e.g., when a user revokes their consent to personalisation), as each request from the shop frontend includes a binary flag denoting the status of the user consent.

5 EXPERIMENTAL EVALUATION

In the following, we first evaluate the prediction quality and index design of VMIS-kNN in Section 5.1, and subsequently evaluate the scalability and business performance of Serenade in offline experiments and an online A/B test (Section 5.2). We provide the code for our experiments at https://github.com/bolcom/serenade-experiments-sigmod.

Datasets. We leverage a combination of public and proprietary click datasets from e-commerce for our offline experiments. We experiment with the publicly available [5] datasets *retailrocket* (an e-commerce dataset from the company "Retail Rocket") and *rsc15* (a dataset used in the 2015 ACM RecSys Challenge), which are commonly used in comparative studies on session-based recommendation [30]. In addition, we create the non-public datasets *ecom-1m*, *ecom-60m*, *ecom-90m* and *ecom-180m* by sampling data from our e-commerce platform with increasing numbers of clicks. The statistics of these datasets are shown in Table 1. Each dataset consists of tuples denoting the `session_id`, `item_id` and `timestamp` of a click event on the platform.

Our proprietary dataset *ecom-180m* is more than six times larger than the largest publicly available dataset *rsc15*. We additionally show statistics of the distribution of clicks per session in the form of its 25th, 50th, 75th and 99th percentile. We find that the majority of sessions on e-commerce platforms is very short (e.g., the median number of clicks per session is less than five) and that these statistics are very similar across all six datasets. In the tail, the sessions from our platform are about twice as long though compared to the public datasets (e.g., the 99th percentile is around 38 clicks in our data and 19 clicks in the public datasets).

	retailr	*rsc15*	*ecom-1m*	*ecom-60m*	*ecom-90m*	*ecom-180m*
clicks	86,635	31,708,461	1,152,438	67,017,367	89,883,761	189,317,506
sessions	23,318	7,981,581	214,490	10,679,757	13,799,762	28,824,487
items	21,276	37,483	110,988	1,760,602	2,263,670	3,305,412
days	10	181	30	29	91	91
public?	yes	yes	no	no	no	no
clicks per session						
p25	2	2	2	2	2	2
p50	2	3	4	4	4	4
p75	4	4	6	7	7	7
p99	19	19	28	36	38	39

Table 1: Public and proprietary datasets for evaluation.

5.1 VMIS-kNN

5.1.1 State-of-the-Art Prediction Quality. Before evaluating systems-related aspects, we run a sanity check experiment for the predictive performance of VMIS-kNN. We aim to confirm that VMIS-kNN also outperforms current neural-network based approaches in the task of session-based recommendation in e-commerce (as recently shown for VS-kNN [24, 30]).

Experimental setup. We replicate the setup from [24, 30], and compare the predictive performance of VMIS-kNN against three recent neural network-based approaches to session-based recommendation (GRU4Rec [20], NARM [27] and STAMP [29]) on various clickstream datasets sampled from our e-commerce platform. We create five versions of the *ecom-1m* dataset by sampling a million clicks from certain months in the past as historical sessions, and measure the prediction quality of the top 20 recommended items for each session of the subsequent day.

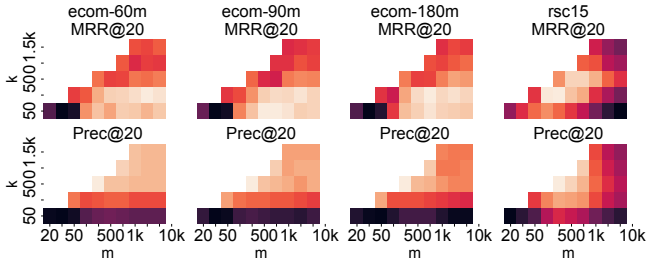


Figure 2: Sensitivity of MRR@20 and Prec@20 to the hyperparameters k (the number of neighbors) and m (the number of most recent sessions per item) in our proprietary datasets.

We optimise the hyperparameters of each approach on samples of the training data, and report the average for each metric over all our evaluation datasets. We report the metric values averaged over all five versions of *ecom-1m*.

Results and discussion. We first investigate the Mean Average Precision (MAP@20), Precision (Prec@20) and Recall (R@20), which denote to what extent an approach correctly predicts the next items in a session. VMIS-kNN outperforms the neural approaches in all of these metrics. Its MAP@20 is .0268 compared to .0251 for the best performing neural approach (GRU4Rec); the Prec@20 of VMIS-kNN is .0722 compared to .0680 for the best performing neural approach (NARM in this case); and VMIS-kNN's R@20 is .378 compared to .359 for the best performing neural approach GRU4Rec. We additionally look at the Mean Reciprocal Rank (MRR@20), which puts a stronger weight on the immediate next item in a session. Again, VMIS-kNN outperforms all neural-based approaches with an MRR@20 of .286 compared to .255 for the best performing neural method (GRU4Rec in this case).

In summary, we confirm that the findings from recent studies on the state-of-the-art performance of VS-kNN also hold for VMIS-kNN on our proprietary data. It is an open question, why neural networks do not outperform conceptually simpler methods in sequential recommendation. There is recent evidence that neural networks have difficulties capturing item frequency information [21], and that many researchers do not adequately compare their proposed neural methods against simple baselines [28, 30].

5.1.2 Sensitivity to Hyperparameter Choices. Next, we investigate the sensitivity of VMIS-kNN to its hyperparameters: the number of neighbors k and the number of most recent sessions per item m.

Experimental setup. We run an exhaustive grid search over 55 combinations of the hyperparameters (the k most similar sessions out of the m most recent sessions) for our four large datasets *ecom-60m*, *ecom-90m*, *ecom-180m* and *rsc15*, where we use the last day as held-out test set.

Results and discussion. Figure 2 illustrates the results of the grid search for MRR@20 and Prec@20 on our datasets with a heatmap where lighter colors indicate better metric values. We observe a unimodal distribution of the resulting metric values for each dataset and metric. The results differ (*i*) based on dataset, e.g., all samples from our proprietary data show similar outcomes, while the distribution for *rsc15* is very different; and (*ii*) based on metric, e.g.,

hyperparameters that work well for MRR (which focuses on the position of the first correctly predicted relevant item) do not necessary provide the best performance for Precision (which considers all correctly predicted relevant items). Our results indicate that VMIS-kNN is easy to tune via offline grid search for a given dataset and target metric.

5.1.3 Index Design. Next, we run a microbenchmark comparing VMIS-kNN vs VS-kNN to validate the performance of our index-based similarity computation.

Experimental setup. We experiment with our index and similarity computation (refered to as *VMIS-kNN*) from Section 3 and compare it against two baseline implementations: (*i*) *VS-kNN* - a baseline implementation that mimics VS-kNN's similarity computation by holding the historical data in hashmaps, and first identifying the m most recent sessions with at least one shared item before computing the similarities, and (*ii*) *VMIS-kNN-no-opt*, a basic variant of VMIS-kNN, which does not contain several optimisations such as early stopping or using octonary heaps instead of binary heaps.

We conduct a micro-benchmark on the *ecom-1m* dataset. We ask each variant to compute the k closest sessions for the sessions from the test set, and we randomly pick the number of items (e.g., the session length) for each session to include in the computation. We repeat this experiment ten times for various values of m (the number of most recent sessions to consider) with six threads, and measure the execution times for $k = 100$ (trying other values of k did not significantly change the results). We implement all algorithms in Rust 1.54 and run the comparison on a machine with an i9-10900KF CPU @ 3.7GHz with ten cores and 64GB of RAM, running Windows 10 21H1.

Results and discussion. The bottom plot in Figure 3(a) shows the resulting runtimes in microseconds for each of our variants. The results are consistent across all values of m: We find that both *VMIS-kNN* and *VMIS-kNN-no-opt* drastically outperform the *VS-kNN* baseline by a factor of three to five. We attribute this observation to the optimised access patterns in the index of VMIS-kNN, which allows us to avoid costly set intersection operations, and the minimisation of intermediate results with our heap data structures (Section 3). We furthermore observe that *VMIS-kNN* consistently outperforms *VMIS-kNN-no-opt* by 6% to 12% which validates our micro optimisations such as early stopping and leveraging octonary heaps instead of binary heaps.

5.2 Serenade

Next, we evaluate our Serenade system. We validate our implementation choices (Section 5.2.1), run a load test for the system in Section 5.2.2, and finally present the results from a three week long A/B test on the live platform (Section 5.2.3).

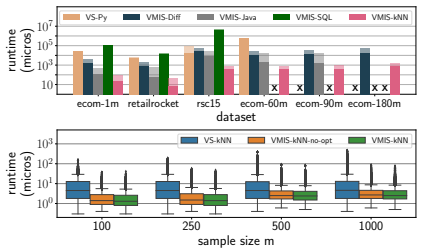
5.2.1 Validation of Implementation Choices. We present an offline experiment which focuses on the performance of our index-based VMIS-kNN approach. We compare our Rust-based implementation against implementations in other programming languages and computational engines to validate our design choice of a custom implementation in Rust. Note that we provide the source code for the alternative implementations in our experiment repository as well.

Experimental setup. We compare our Rust-based VMIS-kNN implementation against four other implementations:

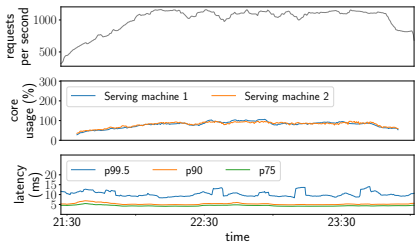
- **VS-Py** – a Python-based implementation of the original VS-kNN approach, based on the reference code [7] from the original VS-kNN paper; we expect this variant to be non-competitive as it is a mere research implementation;
- **VMIS-Diff** – an implementation of VMIS-kNN in *Differential Dataflow* [32], which computes the recommendations incrementally via joins and aggregations; this variant allows us to evaluate the benefits of an incremental similarity computation for growing sessions;
- **VMIS-Java** – an implementation of VMIS-kNN in Java, which stores the historical session data in Java hashmaps; the purpose of this variant is to evaluate the effects of not having full control over the memory management during the similarity computation (and instead relying on a garbage collector);
- **VMIS-SQL** – an implementation of VMIS-kNN in SQL, which leverages the embeddable analytical database engine *DuckDB* [36] in version 0.2.2; the purpose of this variant is to evaluate whether a custom implementation of the approach is necessary; we note that we found it very difficult to express the similarity computation in plain SQL, as it required several deeply nested subqueries;

We ensure through evaluations on held-out data that all variants are correctly implemented and provide equal predictive performance. We expose the historical session data from our public and proprietary datasets to each of these baselines. Next, we ask each implementation to sequentially compute next-item recommendations with a single thread for the growing evolving sessions in the test set of each dataset, and measure the prediction time in microseconds. We run each implementation on a `n1-highmem-8` instance in the Google cloud with 50 gigabytes of RAM, and use $m = 5000$ and $k = 100$ as hyperparameter settings.

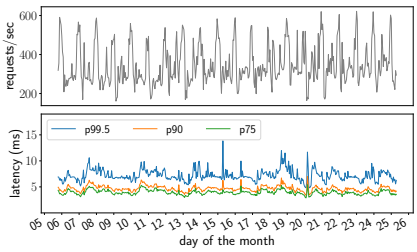
Results and discussion. The top plot of Figure 3(a) illustrates the resulting runtimes from our experiment for the different datasets and baseline implementations. Note that we plot the median and 90th percentile (p90) of these runtimes on a logarithmic scale in a single bar, where the lighter top part denotes the 90th percentile runtime. Our VMIS-kNN implementation consistently outperforms all the baselines both in terms of median and p90 runtime; it is more than two orders of magnitude faster than the Python reference implementation, and more than one order of magnitude faster than the differential dataflow implementation. The second-best implementation is the Java baseline, which is still outperformed by an order of magnitude for the 90th percentile runtime on all datasets except the small *ecom-1m* dataset. When we look at the results for larger datasets, we additionally observe that several baselines start to encounter memory issues (even though they can use 50 gigabytes of RAM), and fail to complete the computation. This happens for the Python implementation (which relies on pandas dataframes internally), for the SQL implementation as well as for the Java variant. Note that our Serenade implementation provides a p90 runtime of at most 1.7 milliseconds on all datasets. We attribute this to the fact that our implementation allows us to carefully control memory allocation and to avoid the materialisation of large intermediate results (such as the complete set of item matches with the historical sessions). We observe that the differential implementation always



(a) (Top) Median and 90th percentile of the computation time per session in microseconds (log-scale) for different VMIS-kNN implementations; (Bottom) Microbenchmark runtimes in microseconds (log-scale) for VMIS-kNN vs. VS-kNN on the *ecom-1m* dataset with k = 100.



(b) Requests per second, core usage in percent and response latency during a load test with more than 1,000 requests per second. Serenade handles about 500 requests per second per core with a 90th percentile latency of less than seven milliseconds.



(c) Requests per second and response latency per hour during our three week long A/B test on the live platform. Serenade responds within less than seven milliseconds in 90% of the cases, even for peak times with more than 600 requests per second.

Figure 3: Offline and online performance of Serenade.

manages to compute results; however, the incremental computation does not pay off runtime-wise, because differential dataflow has to index all intermediate results due to its support for updates in response to input data changes (which is not required in our use case). Finally, we find the SQL implementation to be non-competitive and to not scale to large datasets, which we attribute to the large intermediates from the nested subqueries, and which confirms that a custom implementation of VMIS-kNN is more suitable to scale to large datasets.

5.2.2 Offline Load Test. We finally run a load test in our staging environment to validate that Serenade is able to handle peak production workloads.

Experimental setup. We leverage a setup that resembles our production environment: Serenade's index is built from the last 180 days of browsing activities, covering 6.5 million distinct items. We deploy Serenade on two Kubernetes pods, running on shared core `n1-standard-16` instances in the Google Cloud, where each pod gets provisioned with three cores from an Intel Xeon CPU @ 2.00GHz and 16 GB of RAM.

We generate a simulated load of more than 1,000 requests per second by replaying historical traffic via a load generator application for several hours. We measure the response latency of Serenade as well as the core usage on the machines.

Results and discussion. Figure 3(b) plots the resulting response latency and core usage for our load test. We find that Serenade gracefully handles the load of more than 1,000 requests per second, and responds within less than 7 milliseconds in 90 percent of the cases (`p90`) and in less than 15 milliseconds in 99.5% of the cases (`p99.5`). Each instance uses only one of the three provisioned cores for most of the time. We base our production experiments in the following on the outcomes of this load test.

5.2.3 Online Evaluation in an A/B Test. We present results from a three week long online A/B test on our e-commerce platform, where we compared two variants of Serenade against our existing legacy recommendation system (refered to as *legacy*), which applies a variant of classic item-to-item collaborative filtering [39].

Experimental setup. We show Serenade's recommendations on the product detail page of our e-commerce platform, in a slot titled 'other customers also viewed'. We evaluate two different variants of Serenade: the first variant *serenade-hist* leverages the last two items from each evolving session to compute predictions, while the second variant *serenade-recent* only leverages the most recent item. We set the hyperparameters of VS-kNN to $m = 500$ and $k = 500$, which provide a reasonable trade-off between prediction quality in offline experiments and index size. We run the test for 21 days, in which more than 45 million randomly assigned user sessions were subjected to the recommendations from our variants. We ensure that both the legacy system and Serenade consume the same click data as input at the same time (once per night). Serenade builds its index from the last 180 days of data; after filtering, its daily training data consists of around 111 million sessions with 582 million distinct user-item interactions and contains 6.5M distinct items. We measure the request load to the recommendation system, the response latency (as experienced from the shop frontend) and several business-specific engagement metrics.

Results and discussion. We discuss the systems- and business-specific outcomes of our A/B test.

Response latency. Our most important systems-related metrics is the response latency. Our recommendation systems have to adhere to a strict SLA of responding in less than 50 milliseconds, otherwise requests would be discarded. Recent research also indicates that fast response times help with the acceptance of recommendations in general [24]. Our system architecture and implementation decisions (Section 4) are tailored to allow for low latency responses of our system. This is confirmed by the experimental results illustrated in Figure 3(c), which plots different percentiles of the response latency distribution over the three weeks of our A/B test, and shows the load of the system (in terms of the number of requests per second) for comparison. The request load varies between 200 and 600 requests per second over the day. We find that Serenade's response latencies are very low, the 90th percentile is consistenly around 5 milliseconds, and even the 99.5th percentile is below 10 milliseconds in the majority of cases. This confirms that Serenade exhibits a consistently fast, and stable low-latency response behavior.

CPU usage and operational cost. We deploy Serenade analogously to the setup from the load test in Section 5.2.2: We leverage two Kubernetes pods, running on shared core `n1-standard-16` instances in the Google Cloud, where each pod gets provisioned with cores of an Intel Xeon CPU @ 2.00GHz and 16 GB of RAM. Even with such low resources, Serenade is able to gracefully handle the request workload. We reconfirm the findings from our load test (Section 5.2.2), as Serenade only exhibits a core usage of less than 36% (less than one core) in cases with over 500 requests per second. We also observe a well-behaved linear scaling (with a gentle slope) of the core usage with the number of requests per second.

Customer engagement. Systems-related metrics are important for successfully operating a recommender system, however in the end the recommender system has to perform well in business-related metrics to be valuable for an e-commerce platform. As VMIS-kNN outperforms other approaches in offline evaluations (Section 5.1.1), we are interested to determine how this behavior translates to customer engagement in our A/B test. For that, we measure a conversion-related business metric for the engagement with recommendations on the product detail page.

We find that our session-based recommenders drastically increase this engagement metric for the slot on the product detail page. *Serenade-hist* exhibits a 2.85% increase in the business metric (compared to *legacy*), and *serenade-recent* even shows an increase of 5.72% (both findings are statistically significant). When we control for the overall impact on a site-wide level however, we find that *serenade-recent* exhibits a cannibalising behavior, as it drives down the engagement of other slots on the product detail page (e.g., the 'often bought together' slot). We do not observe this effect for *serenade-hist* though, rendering it the preferred variant.

Summary. We find that Serenade easily handles the load of up to 600 requests per second during our A/B test and consistently generates its recommendations with very low response latency (less than seven milliseconds in the 90th percentile). We furthermore find that the session-based recommendations produced by VMIS-kNN significantly increase customer engagement compared to classical item-to-item recommendations (as produced by our *legacy* system). We would like to highlight that, to the best of our knowledge, we are the first to provide empirical evidence that the superior offline performance of VS-kNN/VMIS-kNN also translates to superior performance in terms of business metrics in a live, real recommender system. This is often not the case for academic recommendation approaches, the winning solution of the highly popularised Netflix prize, for example, never went into production [8].

6 RELATED WORK

Research on recommender systems [11, 12, 17–19, 31, 35, 38, 42, 46, 48–50] is a growing field, with a close connection to industry use cases [43–45], as illustrated by the famous "Netflix Prize" competition [25]. Translating academic progress into deployable solutions has proven to be very difficult though [24], exemplified by the fact that the winning solution of the Netflix prize never went into production [8]. Nearest neighbor-based recommendations, which are in the focus of our work, are a classical approach to recommendation mining [10, 26, 39–41], and are widely deployed in industry [13–15, 22, 34]. Despite their popularity, these approaches are typically outperformed by matrix factorisation- and deep learning-based methods in offline evaluations on classical collaborative filtering problems [25].

However, recent research indicates that nearest neighbor-based approaches provide state-of-the-art performance and outperform neural networks in sequence-based recommendation tasks. An example for such a task is session-based recommendation, which is in the focus of our work, where recent studies [23, 24, 30] indicate that nearest neighbor-based methods outperform previously proposed neural networks [20, 27, 29, 47]. Similar results have been obtained for the more general sequence-based recommendation task of next basket recommendation (where the set of items in a future shopping basket has to be predicted). Here, the nearest neighbor-based state-of-the-art approach TIFU-kNN [21] and simple popularity-based approaches [28] outperform neural networks as well.

7 LEARNINGS & CONCLUSION

We presented our nearest neighbor approach VMIS-kNN as well as the design and implementation of our scalable session-based recommender system *Serenade*. We conducted an extensive offline evaluation of VMIS-kNN and Serenade to validate our design decisions, and detailed results on the latency, throughput and predictive performance of our recommender system from an online A/B test with up to 600 requests per second for 6.5 million distinct items on more than 45 million user sessions on bol.com's e-commerce platform.

In addition to the contributions listed in Section 1, we would like to highlight Serenade's low operational cost: We run two instances with three cores each in the Google cloud (provisioned on shared core `n1-standard-16` instances) for the serving pods, and require 40 minutes on 75 machines of type `n1-highmem-8` for creating the index with Spark every day, which results in a total operational cost of less than 30 euros per day for Serenade. As discussed in Section 5.2.3, Serenade only leverages one of the three cores on each instance, and we only provision the other cores to be prepared for peak loads, e.g., during denial-of-service attacks.

This low cost becomes especially attractive when we compare it with the high cost to train deep learning models. As an example, a neural learning-to-rank model on our platform incurs at least an order of magnitude more cost to be operated on a daily basis, and additionally requires GPU machines for training, which are often a contested resource in the cloud.

In future work, we intend to explore whether we can run our similarity computations on a compressed version of the index, and whether we can incrementally maintain the index with a system such as Differential Dataflow [32].

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

[1] 2021. Actix Web. https://actix.rs.

[2] 2021. d-ary heap. https://docs.rs/dary_heap/0.3.0/dary_heap/.

[3] 2021. Istio sidecars. https://istio.io/latest/docs/reference/config/networking/sidecar/.

[4] 2021. Kubernetes networking services. https://kubernetes.io/docs/concepts/services-networking/service/.

[5] 2021. Performance Comparison of Neural and Non-Neural Approaches to Session-based Recommendation - Additional Information. https://rn5l.github.io/session-rec/.

[6] 2021. RocksDB. https://rocksdb.org.

[7] 2021. VS-kNN reference implementation. https://github.com/rn5l/session-rec/blob/master/algorithms/knn/vsknn.py.

[8] Xavier Amatriain. 2012. Building Industrial-scale Real-world Recommender Systems. *RecSys* (2012), 7–8.

[9] Ioannis Arapakis, Xiao Bai, and B Barla Cambazoglu. 2014. Impact of response latency on user behavior in web search. In *Proceedings of the 37th international ACM SIGIR conference on Research & development in information retrieval*. 103–112.

[10] Badrish Chandramouli, Justin J Levandoski, Ahmed Eldawy, and Mohamed F Mokbel. 2011. StreamRec: a real-time recommender system. *SIGMOD* (2011), 1243–1246.

[11] Tong Chen, Hongzhi Yin, Hongxu Chen, Rui Yan, Quoc Viet Hung Nguyen, and Xue Li. 2019. Air: Attentional intention-aware recommender systems. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 304–315.

[12] Kyung-Jae Cho, Yeon-Chang Lee, Kyungsik Han, Jaeho Choi, and Sang-Wook Kim. 2019. No, that's not my feedback: TV show recommendation using watchable interval. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 316–327.

[13] Abhinandan S Das, Mayur Datar, Ashutosh Garg, and Shyam Rajaram. 2007. Google news personalization: scalable online collaborative filtering. *WWW* (2007), 271–280.

[14] James Davidson, Benjamin Liebald, Junning Liu, Palash Nandy, Taylor Van Vleet, Ullas Gargi, Sujoy Gupta, Yu He, Mike Lambert, Blake Livingston, and Dasarathi Sampath. 2010. The YouTube video recommendation system. *RecSys* (2010), 293–296.

[15] Ted Dunning and Ellen Friedman. 2014. *Practical Machine Learning: Innovations in Recommendation.* " O'Reilly Media, Inc.".

[16] Mohamed Y Eltabakh, Yuanyuan Tian, Fatma Özcan, Rainer Gemulla, Aljoscha Krettek, and John McPherson. 2011. CoHadoop: flexible data placement and its exploitation in Hadoop. *Proceedings of the VLDB Endowment* 4, 9 (2011), 575–585.

[17] Chen Gao, Xiangnan He, Dahua Gan, Xiangning Chen, Fuli Feng, Yong Li, Tat-Seng Chua, and Depeng Jin. 2019. Neural multi-task recommendation from multi-behavior data. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 1554–1557.

[18] Lei Guo, Hongzhi Yin, Qinyong Wang, Bin Cui, Zi Huang, and Lizhen Cui. 2020. Group recommendation with latent voting mechanism. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 121–132.

[19] Jiayuan He, Jianzhong Qi, and Kotagiri Ramamohanarao. 2019. A joint context-aware embedding for trip recommendations. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 292–303.

[20] Balázs Hidasi, Alexandros Karatzoglou, Linas Baltrunas, and Domonkos Tikk. 2015. Session-based recommendations with recurrent neural networks. *arXiv:1511.06939* (2015).

[21] Haoji Hu, Xiangnan He, Jinyang Gao, and Zhi-Li Zhang. 2020. Modeling personalized item frequency information for next-basket recommendation. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*. 1071–1080.

[22] Yanxiang Huang, Bin Cui, Wenyu Zhang, Jie Jiang, and Ying Xu. 2015. TencentRec: Real-time Stream Recommendation in Practice. *SIGMOD* (2015), 227–238.

[23] Dietmar Jannach and Malte Ludewig. 2017. When recurrent neural networks meet the neighborhood for session-based recommendation. *RecSys* (2017), 306–310.

[24] Barrie Kersbergen and Sebastian Schelter. 2021. Learnings from a Retail Recommendation System on Billions of Interactions at bol.com. *ICDE* (2021).

[25] Yehuda Koren, Robert Bell, and Chris Volinsky. 2009. Matrix factorization techniques for recommender systems. *Computer* 42, 8 (2009).

[26] Justin J Levandoski, Mohamed Sarwat, Mohamed F Mokbel, and Michael D Ekstrand. 2012. RecStore: an extensible and adaptive framework for online recommender queries inside the database engine. *EDBT* (2012), 86–96.

[27] Jing Li, Pengjie Ren, Zhumin Chen, Zhaochun Ren, Tao Lian, and Jun Ma. 2017. Neural attentive session-based recommendation. *CIKM* (2017), 1419–1428.

[28] Ming Li, Sami Jullien, Mozhdeh Ariannezhad, and Maarten de Rijke. 2021. A Next Basket Recommendation Reality Check. *arXiv preprint arXiv:2109.14233* (2021).

[29] Qiao Liu, Yifu Zeng, Refuoe Mokhosi, and Haibin Zhang. 2018. STAMP: short-term attention/memory priority model for session-based recommendation. *KDD* (2018), 1831–1839.

[30] Malte Ludewig, Noemi Mauro, Sara Latifi, and Dietmar Jannach. 2019. Performance comparison of neural and non-neural approaches to session-based recommendation. In *Proceedings of the 13th ACM Conference on Recommender Systems*. 462–466.

[31] Sreekanth Madisetty. 2019. Event recommendation using social media. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 2106–2110.

[32] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. 2013. Differential Dataflow.. In *CIDR*.

[33] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. 2016. Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research* 17, 1 (2016), 1235–1241.

[34] Sean Owen. 2012. *Mahout in action*. Vol. 10. Manning Shelter Island, NY.

[35] Andreas Pfadler, Huan Zhao, Jizhe Wang, Lifeng Wang, Pipei Huang, and Dik Lun Lee. 2020. Billion-scale Recommendation with Heterogeneous Side Information at Taobao. In *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*. IEEE, 1667–1676. https://doi.org/10.1109/ICDE48307.2020.00148

[36] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an embeddable analytical database. In *Proceedings of the 2019 International Conference on Management of Data*. 1981–1984.

[37] Pengjie Ren, Zhumin Chen, Jing Li, Zhaochun Ren, Jun Ma, and Maarten De Rijke. 2019. Repeatnet: A repeat aware neural recommendation machine for session-based recommendation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 4806–4813.

[38] Francesco Ricci, Lior Rokach, and Bracha Shapira. 2011. Introduction to recommender systems. In *Recommender systems handbook*. 1–35.

[39] Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl. 2001. Item-based collaborative filtering recommendation algorithms. In *Proceedings of the 10th international conference on World Wide Web*. 285–295.

[40] Sebastian Schelter, Christoph Boden, and Volker Markl. 2012. Scalable similarity-based neighborhood methods with mapreduce. *RecSys* (2012), 163–170.

[41] Sebastian Schelter, Ufuk Celebi, and Ted Dunning. 2019. Efficient incremental cooccurrence analysis for item-based collaborative filtering. In *Proceedings of the 31st International Conference on Scientific and Statistical Database Management*. 61–72.

[42] Junshuai Song, Zhao Li, Zehong Hu, Yucheng Wu, Zhenpeng Li, Jian Li, and Jun Gao. 2020. Poisonrec: an adaptive data poisoning framework for attacking black-box recommender systems. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 157–168.

[43] Manos Tsagkias, Tracy Holloway King, Surya Kallumadi, Vanessa Murdock, and Maarten de Rijke. 2021. Challenges and research opportunities in ecommerce search and recommendations. In *ACM SIGIR Forum*, Vol. 54. ACM New York, NY, USA, 1–23.

[44] Chi-Man Wong, Fan Feng, Wen Zhang, Chi-Man Vong, Hui Chen, Yichi Zhang, Peng He, Huan Chen, Kun Zhao, and Huajun Chen. 2021. Improving Conversational Recommendation System by Pretraining on Billions Scale of Knowledge Graph. *ICDE* (2021).

[45] Xu Xie, Fei Sun, Xiaoyong Yang, Zhao Yang, Jinyang Gao, Wenwu Ou, and Bin Cui. 2021. Explore User Neighborhood for Real-time E-commerce Recommendation. *ICDE* (2021).

[46] Hongzhi Yin, Qinyong Wang, Kai Zheng, Zhixu Li, Jiali Yang, and Xiaofang Zhou. 2019. Social influence-based group representation learning for group recommendation. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 566–577.

[47] Fajie Yuan, Alexandros Karatzoglou, Ioannis Arapakis, Joemon M Jose, and Xiangnan He. 2019. A simple convolutional generative network for next item recommendation. *WSDM* (2019), 582–590.

[48] Yu Zheng, Chen Gao, Xiangnan He, Yong Li, and Depeng Jin. 2020. Price-aware recommendation with graph convolutional networks. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 133–144.

[49] Xiangmin Zhou, Dong Qin, Xiaolu Lu, Lei Chen, and Yanchun Zhang. 2019. Online social media recommendation over streams. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 938–949.

[50] Zainab Zolaktaf, Reza Babanezhad, and Rachel Pottinger. 2018. A generic top-n recommendation framework for trading-off accuracy, novelty, and coverage. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 149–160.