ROSE: Robust Caches for Amazon Product Search

Chen Luo, Vihan Lakshman, Anshumali Shrivastava, Tianyu Cao, Sreyashi Nag, Rahul Goutam, Hanqing Lu, Yiwei Song, Bing Yin
Amazon Search
Palo Alto, California, USA

ABSTRACT
Product search engines like Amazon Search often use caches to improve the customer user experience; caches can improve both the system’s latency as well as search quality. However, as search traffic increases over time, the cache’s ever-growing size can diminish the overall system performance. Furthermore, typos, misspellings, and redundancy widely witnessed in real-world product search queries can cause unnecessary cache misses, reducing the cache’s utility. In this paper, we introduce ROSE, a RObuSt cachE, a system that is tolerant to misspellings and typos while retaining the look-up cost of traditional caches. The core component of ROSE is a randomized hashing schema that makes ROSE able to index and retrieve an arbitrarily large set of queries with constant memory and constant time. ROSE is also robust to any query intent, typos, and grammatical errors with theoretical guarantees. Extensive experiments on real-world datasets demonstrate the effectiveness and efficiency of ROSE. ROSE is deployed in the Amazon Search Engine and produced a significant improvement over the existing solutions across several key business metrics.

CCS CONCEPTS
• Information systems → Query log analysis; Query intent; Query reformulation.

KEYWORDS
Amazon Search, Robust Cache, Data Mining

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ACM Reference Format:

1 INTRODUCTION
Online shopping has become an essential part of consumers’ daily lives in recent years and has seen a dramatic increase in demand during the ongoing COVID-19 global pandemic. As a critical component of an e-commerce website, the product search engine connects the customer intent with the product selections. Improving the product search engine’s performance is critical to a better shopping experience. Two key factors impact the search engine’s performance: (1) The response time to a customer request and (2) Providing high-quality results that match the customers’ intent.

User studies show that slow responses cause perceived interruptions to the shopping experience and even site abandonment. The response time is also a key factor for the product search engine’s throughput planning. Modern product search engines are usually composed of different expensive machine learning models [1, 6, 7, 11, 14, 15, 22, 30, 31, 33], such as relevance matching models [20], ranking models [2], and query annotation models [29]. Serving the entirety of search traffic through expensive deep learning models is prohibitive in real-world product search engines due to latency limitations, and cost considerations [12]. Thus, instead of serving all queries through these expensive deep learning models, a more practical solution is to serve frequent queries from a cache.

However, traditional caches suffer from the trade-off between the cache miss rate and the cache size. Having a small cache size will lead to a high cache miss rate. On the other hand, as product search engines scale, the set of frequently occurring queries becomes prohibitively large, and grows due to morphological variants of queries with the same intent. For instance, "Nike shoes", "Nike shoe", and "Nike's shoe" may all be cached queries due to their frequency. These queries all share the same intent, and they artificially inflate the cache size and diminish performance. Therefore, designing a robust cache that is invariant to typos and morphological differences is critical for scaling real-world search services since it enables increasing the cache hit rate without correspondingly increasing the latency and memory footprint.

Moreover, one key issue that hurts the quality of search results is the presence of low-performing queries, which are queries for which the search engine fails to return high-quality results. Analysts show that most of these failure cases are due to typographical
Figure 2: The overall framework of ROSE. ROSE contains two phases: (1) Cache Index Generation: generating the robust index using the input queries. (2) Online Retrieval: mapping the input query to one of the queries in the cache.

errors [26]. These low-performing queries are usually lexically or semantically similar to some frequently searched, well-performing queries that produce satisfactory results. Thus, if we could map these low-performing queries to a frequently searched query with the same intent via a robust caching mechanism, we would be able to improve search quality. Furthermore, this query mapping process would also reduce latency since product search engines typically cache these frequently issued queries and their corresponding behavioral information for faster serving, as in Fig 1.

To solve these challenges, we propose ROSE to cache well-performing or frequent queries to improve the response time and search quality of the product search engine. The core component of ROSE is a randomized hashing structure that indexes the query set while preserving the lexical or semantic information. Specifically, our paper includes the following contributions:

- **Operational System**: We introduce ROSE, a comprehensive end-to-end solution for caching queries for product search. ROSE can index and perform look-ups on web-scale data in constant time and constant memory and is faster than other alternatives by orders of magnitude.

- **Technical Novelty**: We invented a system that combines multiple powerful randomized algorithmic techniques, including locality sensitive hashing, reservoir sampling, and count-based k-selection, in a novel way that together allow us to scale up ROSE to massive query sets while maintaining constant-time retrieval.

- **Real-World Impact**: We deployed ROSE in the Amazon product search engine, showing improvements in system performance and business metrics when compared to the existing solution.

## 2 ROSE: ROBUST CACHE VIA RANDOMIZED HASHING

In this section, we describe ROSE, a robust cache for queries via randomized hashing. ROSE contains two phases, Index Generation and Online Retrieval, as illustrated in Fig 2. We first introduce these two phases, followed by a theoretical analysis of ROSE in terms of both time and memory complexity.

### 2.1 ROSE Index Generation

We design the index generation process of ROSE under two requirements. First, the cache needs to capture the query similarity, meaning that cache needs to take the similarity of queries into account when performing look-ups to be robust to typos and semantic variance. Secondly, due to the large-scale indexing space of real product search engines, the cache size needs to avoid scaling with the volume of queries.

To capture the textual similarity information, we use locality-sensitive hashing (LSH) [8] for the index generation phase. LSH generates signatures for input data under a certain similarity measure. The signatures generated by LSH capture the similarity information between queries such that similar queries have a high probability of having the same hashing signature and thus colliding. Since LSH is a randomized procedure, we boost the probability of hashing similar queries together by maintaining L independent hash tables for our index. This work shall focus on two hashing strategies: lexical preserving hashing and product type preserving hashing. We will introduce the details of these two hash functions in Section 2.3 and Section 2.4, respectively.

However, under the locality-sensitive hashing framework, the size of the hash tables increases linearly with the volume of data [25] which leads to an explosion in memory footprint when working with web-scale data. To solve this problem, inspired by the work in [28], we use a reservoir sampling strategy to fix our cache’s memory usage and preserve the data’s similarity information.

The reservoir sampling algorithm [27] processes a stream of m numbers and generates R uniform samples by only using an array of size R, where R ≪ m. Moreover, reservoir sampling only needs one pass over the data, and does not increase the computational complexity of the index generation process. We will provide a theoretical analysis of this sampling strategy as applied in our caching framework in Section 2.5.

### 2.2 ROSE Online Retrieval

Given a search query, we perform a robust cache lookup by first computing the LSH signature of this query and looking up the corresponding bucket in the hash tables. We then rank the similarity of the cached queries within the bucket to the new search and return the top result. However, under the standard LSH schema [16–18], we still have to calculate the pairwise similarities inside the bucket to retrieve the top result, which can be expensive, especially since product search engines typically maintain strict latency budgets.

To avoid this expensive pairwise similarity computation, we use the count-based k-selection strategy inspired by [13]. Across the L different hash tables, we observe that the cached entries with the greatest number of collisions with the new query are more similar to the query. This observation allows us to estimate the actual ranking in an unbiased manner. We count each data point’s frequency of occurrence in the aggregated reservoirs and rank all the data points based on the frequency. By using this strategy, the online retrieval process runs in constant time (Section 2.5).

### 2.3 Lexical Preserving Hashing

Our goal for lexical preserving hashing is to design a hash function that preserves the lexical similarity among input queries. To achieve this in product search, we use the Jaccard similarity to measure the similarity between two queries, defined as the ratio of character spans that two query keywords share, and use minhash [3] as the corresponding LSH scheme.

Given a query $Q$ of $n$ characters and $m$ words, we slice these keywords into a set of subsequences consisting of character-level
sequences and word unigrams, denoted by \( S(Q) = \{ c_i \}_{i=1}^n \cup \{ c_{i+1} \}_{i=1}^{n-1} \cup \{ w_i \}_{i=1}^m \), where \( c_i \) and \( w_i \) denote the \( i \)-th character and word of the query, respectively. The length of the character subsequence is a hyper-parameter. We find that a subsequence length of 3 gave us the best results. We then use the recent advances in densified one permutation hashing (DOPH) [24] to compute the minhash signatures of \( S(Q) \) efficiently.

### 2.4 Product Type Preserving Hashing

In a product search engine, understanding the product type information of a query is crucial to showing relevant results that match the customer’s intent and avoid, for instance, returning dishwasher accessories in response to a search for dishwashers. Thus, when performing a cache lookup, it is critical that we map the original query to one that preserves the original product type intent.

To preserve the product information, we add weights to product type tokens in the query. The product type tokens are extracted by a production NER model [32]. We use the same process as lexical preserving hashing to generate the token set \( S(Q) \) for the input query. We then assign weights to the tokens in \( S(Q) \) by the following strategy: If a token is not a product type token, we give a weight of 1.0. Otherwise, we assign weight \( W > 1 \) to this token. Here, \( W \) is a hyperparameter in our algorithm. In our real-world experiments, we find \( W = 10 \) gave us the best results. To generate the hash signatures of the weighted set \( S(Q) \), we leverage recent advances in efficiently computed weighted minhash signatures [5, 9, 23].

### 2.5 Theoretical Analysis

In this subsection, we analyze the complexity of our algorithm.

**Indexing Step Time Complexity:** In the proposed algorithm, the average time complexity of computing the hashes for one query is \( O(LT) \), where \( L \) is the number of repetitions of LSH and \( T \) is the average number of tokens per query. The complexity of generating the entire robust cache structure is \( O(LNT) \) for a dataset with \( N \) queries. In practice, \( L, T \) and \( T \) are small constants much less than \( N \), so we can consider asymptotic time complexity to be \( O(N) \). This linear time complexity of building the cache gives our method a significant scaling advantage to cache a massive amount of data.

**Retrieval Step Time Complexity:** The time complexity of ROSE’s retrieval step is \( O(LTBL) \). \( O(LT) \) is the complexity of calculating the hash values for the incoming query. \( O(BL) \) is the time complexity of \( k \)-selection in the combined sets, where \( B \) is the bucket size. Therefore, the retrieval step’s overall time complexity is \( O(L^2BT) \), independent of the cache size \( N \). In practice, \( L, B \) and \( T \) are small constants. As a result, cache retrieval’s time complexity is constant, which gives ROSE the decisive advantage for latency-critical services like product search.

**Memory Complexity:** The memory usage of ROSE is \( O(B \cdot NB \cdot L) \), where \( NB \) is the number of buckets in one hash table. \( NB \) is a hyperparameter and is a constant number independent of the cache size. We can see that the memory usage is not increasing with the size of the cache. This enables ROSE to achieve fast retrieval speeds on massive data with minimal memory costs, an ideal combination for industry-scale search engines.

**Error Analysis:** Due to the randomized nature of LSH, we note that it is possible to map the original query to an unrelated bucket with some small, but nonzero probability. However, we can dramatically reduce this error probability by maintaining \( L \) independent hash tables. In particular, we can apply standard Chernoff bounds arguments [19] and conclude that the probability of an error in more than, for instance, half of the \( L \) hash tables decreases exponentially as a function of \( L \).

### 3 OFFLINE EXPERIMENTS

**Dataset:** We sampled approximately 60 million well-performing queries from Amazon search logs as our cache’s target set. Following the same evaluation strategy in [21], our evaluation dataset samples queries from three buckets: a) \( HQ \): Normal Queries, which are those in the top tercile of frequency, b) \( HQ \): Hard queries sampled from the middle tercile of queries by frequency, and c) \( LTQ \): Long-tail queries in the bottom tercile of frequency.

We randomly selected these queries from the search logs over one month. Each of these three sets contains 1000 queries. We obtained the re-mapped results for the queries from various query caching strategies and used a group of highly trained human judges to assign a binary relevance grade (relevant or irrelevant) to each returned query with respect to the original query’s intent. This relevant grade is used for calculating the performance metrics of different methods.

**Experimental Design:** We designed the experiments to answer two critical questions: a) **Robustness:** How accurate is ROSE’s retrieval process? b) **Efficiency:** How efficient is ROSE’s indexing and retrieval process? So, we test the following methods:

- **R-LP:** This method is our proposed method, ROSE, with lexical preserving hashing. The number of hash tables is \( L = 36 \) and the number of hashes is \( K = 3 \).
- **R-PT:** This method is our proposed method, ROSE, with product type preserving hashing. All the other hyperparameters are the same as ROSE-LP.
- **EC:** This is the exact-match cache implemented as a standard hash map. In the retrieval phase, the Exact-only cache returns the exact match candidates.
- **BF:** This is a cache structure designed by replacing ROSE’s retrieval algorithm with brute force search. We use edit distance as our similarity measure, computed via a dynamic programming algorithm

<table>
<thead>
<tr>
<th>Data</th>
<th>Metrics</th>
<th>R-LP</th>
<th>R-PT</th>
<th>EC</th>
<th>BF</th>
<th>FC</th>
</tr>
</thead>
<tbody>
<tr>
<td>NQ</td>
<td>Prec</td>
<td>.88±.03</td>
<td>.96±.01</td>
<td>.10±.00</td>
<td>.90±.02</td>
<td>.96±.08</td>
</tr>
<tr>
<td></td>
<td>Recall</td>
<td>.81±.02</td>
<td>.90±.04</td>
<td>.50±.04</td>
<td>.88±.02</td>
<td>.89±.09</td>
</tr>
<tr>
<td></td>
<td>F1</td>
<td>.84±.05</td>
<td>.93±.08</td>
<td>.70±.04</td>
<td>.89±.09</td>
<td>.92±.03</td>
</tr>
<tr>
<td>HQ</td>
<td>Prec</td>
<td>.78±.01</td>
<td>.90±.03</td>
<td>.10±.00</td>
<td>.80±.03</td>
<td>.89±.07</td>
</tr>
<tr>
<td></td>
<td>Recall</td>
<td>.80±.09</td>
<td>.86±.05</td>
<td>.52±.05</td>
<td>.79±.08</td>
<td>.85±.07</td>
</tr>
<tr>
<td></td>
<td>F1</td>
<td>.79±.06</td>
<td>.88±.09</td>
<td>.39±.06</td>
<td>.79±.08</td>
<td>.87±.07</td>
</tr>
<tr>
<td>LTQ</td>
<td>Prec</td>
<td>.77±.03</td>
<td>.73±.06</td>
<td>1.0±.00</td>
<td>.76±.04</td>
<td>.75±.02</td>
</tr>
<tr>
<td></td>
<td>Recall</td>
<td>.79±.04</td>
<td>.76±.03</td>
<td>.12±.03</td>
<td>.75±.03</td>
<td>.78±.02</td>
</tr>
<tr>
<td></td>
<td>F1</td>
<td>.78±.05</td>
<td>.74±.05</td>
<td>.21±.03</td>
<td>.75±.04</td>
<td>.76±.05</td>
</tr>
<tr>
<td>Indexing Time</td>
<td>65min</td>
<td>75min</td>
<td>10min</td>
<td>0min</td>
<td>120min</td>
<td></td>
</tr>
<tr>
<td>Retrieval Time</td>
<td>1.8ms</td>
<td>2.1ms</td>
<td>1.1ms</td>
<td>0.0ms</td>
<td>65min</td>
<td></td>
</tr>
</tbody>
</table>

### Table 1: Offline Experiment Results

We deployed WWW '22 Companion, April 25–29, 2022, Virtual Event, Lyon, France

Chen Luo et al.

- FC: This is a designed cache structure for embedding vectors by replacing ROSE's indexing and retrieval algorithm with FAISS [10]. We obtain the embeddings for each input query using a semantic product embedding model [20]. We choose the hyperparameters suggested by [10].

We adopted three commonly used metrics for offline evaluation: Precision, Recall, and $F_1$ Measure. To compute these metrics, we utilize human judgments of relevance. We also analyze the speed of different methods for the indexing generation time and the online retrieval time.

**Overall Performance:** The results of all methods under the five metrics are presented in Table 1. Compared with other methods, ROSE performs the best on all three datasets. Specifically, ROSE offers a relative performance gain of 1.2% in Recall and 2.0% in $F_1$ over the best baselines averaged across the three datasets. In particular, we find that the improvements of ROSE-PT on Normal queries and Hard Queries are more significant than Long-tail queries. On the other hand, ROSE-LP performs better on long-tail queries compared to ROSE-PT. Additionally, ROSE not only achieves a superior quality over these competing methods, but does so with much better efficiency. ROSE is significantly faster in terms of index generation time and online retrieval time. In particular, ROSE-LP completes the index generation process in 65 minutes while ROSE-PT requires 73 minutes. Compared with other caches such as BF-Cache and FAISS-Cache, ROSE has a decisive speed advantage. ROSE can finish the online retrieval process in around 2ms, while FAISS-cache needs 120ms and BF-Cache requires 65 minutes. In summary, ROSE shows strong retrieval performance with extremely low latency and minimal cost, which makes it a compelling solution for latency-critical services such as product search engines.

4 **SYSTEM DEPLOYMENT IN AMAZON**

4.1 **ROSE for Query Rewrite**

We deployed ROSE within the Amazon.com product search engine to rewrite problematic user queries, such as those with typos, to alternative queries that provide a better user experience. We refer to this system as ROSE-QR. Leveraging lexical-preserving hashing, ROSE-QR maps an incoming query to one of the existing cached queries that have high-quality results according to lexical similarity.

We ran an online A/B experiment on the Amazon search engine to test ROSE-QR’s impact on the user experience. In the online experiment, users in the treatment group saw expanded search results from the alternative queries generated by ROSE-QR. Professional human judges measured the quality of the top search results shown in each arm of the experiment. We tracked the reduction of recall failures when the search engine does not return enough results for the user queries. We also measured business metrics such as revenue and purchased units. Our system did a better job in providing more relevant results, as measured by human evaluators, and significantly improved several business metrics as shown in Table 2.

4.2 **ROSE for Product Type Annotation**

The intended product type, such as shoes in the query "red nike shoes", is the most critical information in a user query. Identifying

<table>
<thead>
<tr>
<th>ROSE-QR Metric</th>
<th>Gain</th>
<th>Filter Method</th>
<th>Defects Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Revenue</td>
<td>+0.42%</td>
<td>No Filtering</td>
<td>11.1%</td>
</tr>
<tr>
<td>Purchases</td>
<td>+0.30%</td>
<td>ROSE-PT</td>
<td>9.4%</td>
</tr>
<tr>
<td>Click-Through Rate</td>
<td>+7.26%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Production Impact of ROSE in Amazon Search

![Figure 3: The top-4 result for query "dishwasher".](image)

Figure 3: The top-4 result for query "dishwasher". The correct product type from the query helps the search engine retrieve the correct products and display a search result page layout customized for each product type.

We implemented ROSE to cache the intended product type of 5-10 million frequent queries. For an incoming tail query, ROSE maps the query to a few cached queries and uses the retrieved cached product types as the prediction for the tail query’s product type. To evaluate the impact on user experience, we used our ROSE product type prediction model to filter out irrelevant search results with the wrong product types, such as a dress for the query "red nike shoes" as in Fig. 3. We deployed this system in the Amazon.com product search engine and measured the search defect rate with and without product type recognition. We define the product type defect rate as the number of products in the top 16 results with the wrong product type. From Table 2, we observe that, by using ROSE, the defect rate decreased by 1.7%, a significant improvement to the user experience.

5 **CONCLUSION**

In this paper, we present ROSE for product search. ROSE is a robust cache that maps an online query to cached queries by preserving the query intent (lexically or semantically). The proposed model is highly scalable and can deal with hundreds of millions of candidates in constant time and constant memory. We provide both a theoretical analysis of ROSE as well as an extensive offline evaluation. We deployed ROSE in the Amazon.com search engine and witnessed significant improvement over the existing solutions in terms of system performance and business metrics.