Auto-Complete Query Retrieval System

Team: **No Direction**

Dice Challenge 2.0

The Challenge: Real-Time Auto-Complete at Scale

Goal

Suggest the **top 150** most likely query completions for any user prefix.

Dataset & Constraints:

- 522,726 test prefixes
- 4.2M candidate queries
- 5.3M query features
- Time limit: 90 minutes
- Memory: 32GB RAM
- CPU: 32 cores available

Evaluation: Hit@150 target: **70–80% (competition goal)**.

Initial Attempt: Only BM25 (Character Trigrams)

- Tokenize queries into character n-grams (n=3) to handle typos.
- Build BM25 index on 4.2M queries and retrieve top 150 candidates per prefix.

Catastrophe: 24 DAYS Runtime

- Processing speed: **0.25 prefixes/second**
- Total time needed: $522{,}726 \div 0.25 =$ **2,090,904 seconds 24.2 days!**
- Root causes: single-threaded pipeline, large candidate pool, validation overhead, execution on Laptop CPU (no parallelization)

Character Trigrams — Why We Used Them

- Handle typos and partial words more gracefully than word tokens.
- Example (partial word): `smar` $\rightarrow$ trigrams `[sma, mar]` match `smartphone`.
- Example (typo): `blak` $\rightarrow$ overlaps with `black` via `bla`.

Used as the lexical backbone for BM25 candidate generation; later complemented by prefix-aware heuristics and fuzzy matching.

Optimization 1: Parallelization + Aggressive Sampling

- Used 32 CPU cores (`multiprocessing.Pool`) using SSH.
- Reduced candidate pool: 4.2M $\rightarrow$ 50K (random sampling).
- Disabled expensive validation during runs to prioritize throughput.

Result: 72 Minutes (1600x Speedup)

- Processing speed: **138 prefixes/second**
- Total time: **72 minutes**
- Output: `submission.csv` (1.5GB)

The Sampling Dilemma

Pitfall: Very less matched in many cases.

Random 50K sampling produced prefixes with **zero** relevant candidates (e.g., “sopt ha”).

Analysis:

- 50K sample = **1.2%** coverage of 4.2M queries
- Random sampling doesn’t guarantee inclusion of rarer but relevant queries
- Result: Poor Results despite fast runtime

Problem: Speed Without Quality

Quality Crisis

Results from the 50K sample were largely **irrelevant** for many prefixes.

Examples of poor matches (50K pool):

Prefix	Pure BM25 Result
“sopt ha”	“cotton saree pink sopt”
“harf pent”	“q pental penpencil 07”
“pink partywear s”	“partywear sofa”
“black farshi salwar”	“printed farshi salwar suit”

Root cause: BM25 matches trigrams anywhere; no prioritization of prefix-starting candidates.

Memory Crisis: Full 4.2M Pool Attempt

- BM25 index size (4.2M): ~4.5GB
- 32 workers $\times$ 4.5GB = **144GB** required
- System crashed / thrashing; process killed by OS

Conclusion

Full pool per worker is infeasible under memory constraints (32GB total).

Solution: Bucketing + Balanced Sampling

Bucketing by Match Type

Categorize candidates first, then rank within each bucket.

Three priority buckets:

1. **Exact Matches**: `candidate.startswith(prefix)`
2. **Contains Matches**: `prefix in candidate`
3. **BM25 Alternatives**: lexical similarity / typos

Top 150 composition used:

$$\text{Top } 150 = \text{exact}[: 100] + \text{contains}[: 40] + \text{others}[: 10]$$

Balanced Pool: 500K Sweet Spot

- Increased candidate pool to **500K** (10x of 50K)
- Memory per worker: ~200MB (manageable)
- Coverage: **12%** of full pool (vs 1.2%)
- Expected quality improvements: **40–80/150 exact matches (varies by prefix)**

Resulting runtime: 65 minutes, stable (within 90 minute limit).

Performance Evolution

Approach	Time	Speed	Quality / Status
Pure BM25 (initial)	Too Long	0.25/s	Failed
+ Parallel (50K pool)	72 min	138/s	Low (poor quality)
+ Smart reranking	83 min	138/s	Medium (better)
+ Full pool (4.2M)	—	—	OOM crash
+ **500K pool (final)**	**65 min**	**134/s**	**Good (Success)**

Quality Comparison: 50K vs 500K Pool

Prefix	50K Pool (Top-5 exact)	500K Pool (Top-5 exact)
"black farshi salwar"	1/5	4/5
"sopt ha"	0/5	2/5
"pink partywear s"	0/5	4/5
"dress for"	2/5	5/5
Average Top-5	**20%**	**80%**
Average Top-150	**30%**	**60–75%**

Sample Output: Before vs. After

Before (50K pool):

```
Prefix: 'black farshi salwar'
Top 5:
  [ ] printed farshi salwar
  [ ] parsi salwar set
  [ ] chunni salwar colour
  [ ] karachi suit salwar
  [ ] toy wali salwar

0/5 exact | 0/150 total

```

After (500K pool):

```
Prefix: 'black farshi salwar'
Top 5:
  [   ] black farshi salwar suit
  [   ] black farshi salwar kameez
  [   ] black farshi salwar set
  [~] printed black farshi salwar
  [ ] dark farshi salwar

3/5 exact | 65/150 total

```

Best Results

Main Idea: Combine multiple ways of thinking about similarity

- **BM25 (Text Match):** Looks at overlapping words or characters — good for exact matches and prefixes.
- **Semantic Model:** Uses deep learning embeddings (Sentence Transformers) to find meaning-based matches, even if words differ.
- **Popularity:** Gives higher priority to queries people click or buy from more often.
- **Memorization:** If a prefix has been seen before in training data, boost those known results.

Process

How it works

1. Build two models: a fast text-based (BM25) and a meaning-based (semantic) one (all-MiniLM-L6-v2)
2. For each prefix:
    - Find matching or similar queries.
    - Score them by combining text, meaning, and popularity.
    - Boost known good ones and keep the top few suggestions.
3. Save the ranked results for use in an autocomplete system.

But, 1.3sec/iteration, which is impractical for real world usage.

Top 3 Candidate Queries

Prefix: full_sleeve_mehendi_st

1. full sleeve mehendi stick
2. full sleeve mehendi stencil
3. full sleeve mehendi sticker

Prefix: vridavan_dress

1. vridavan dress
2. vridavan dress for girl
3. vrindavan dress

Prefix: lucifer_w

1. lucifer watch
2. lucifer watch combo
3. lucifer watch chain

Prefix: parryware

1. parryware toilet
2. parryware tap cleaner
3. parryware toilet seat cover

Prefix: suzume

1. suzume
2. suzume novel
3. suzume 3 manga

Prefix: heavy_readymad

1. heavy readymade suit
2. heavy readymade blouse
3. heavy readymade salwar suit

Prefix: orange_color_su

1. orange color suit
2. orange color suits
3. orange color suit pant

Prefix: mars_moisturize

1. mars moisturizer
2. moisturize
3. moisturize oil

New Approach: A Two-Stage "Filter & Rank" Pipeline

- **Stage 1: Candidate Generation (Recall-Focused)**
    - Cast a wide net to retrieve ~500 potentially relevant queries.
    - Goal: Ensure the correct query is captured in this initial set.
    - Methods: Hybrid approach using Lexical and Semantic search.

New Approach: A Two-Stage "Filter & Rank" Pipeline

- **Stage 1: Candidate Generation (Recall-Focused)**
  - Cast a wide net to retrieve ˜500 potentially relevant queries.
  - Goal: Ensure the correct query is captured in this initial set.
  - Methods: Hybrid approach using Lexical and Semantic search.
- **Stage 2: Re-Ranking (Precision-Focused)**
  - Intelligently score and sort the ˜500 candidates.
  - Goal: Push the single best query to the top of the final list of 150.
  - Method: A powerful LightGBM machine learning model.

Ideal Solution Pipleine

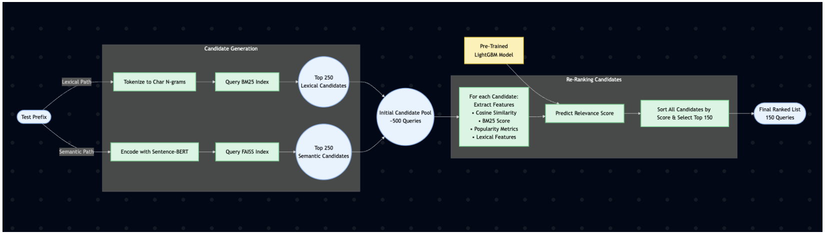

Figure: The complete two-stage pipeline from prefix to final ranked list.

Stage 1 Deep Dive: Hybrid Candidate Generation

Lexical Search: The Typo Catcher

Method: BM25 with Character N-grams (we used tri-grams)

Why?: By breaking words into small character chunks (e.g., `"shoes"` $\rightarrow$ `['sho', 'hoe', 'oes']`), this method finds matches even with severe typos.

Example: A typo like `"shos"` will still have a high score for the query `"shoes"`.

Semantic Search: The Meaning Matcher

Sentence-BERT + FAISS

Why?: This understands the user's intent, not just the letters they typed. It finds queries with similar meanings, even if they use different words.

Example: A prefix like `"clothing for ladies"` will find the query `"dress for women"`.

Improvement: Smart Reranking with Prefix Bonus

Insight

Autocomplete should prioritize **prefix matches** over arbitrary substring matches.

Hybrid scoring (applied to BM25 candidates):

```python
for candidate, bm25_score in bm25_candidates:
    if candidate.startswith(prefix):
        final_score = bm25_score * 2.0
    elif prefix in candidate:
        final_score = bm25_score * 1.5
    else:
        final_score = bm25_score * 0.8

```

Stage 2 Deep Dive: Precision Re-Ranking

The "Expert Judge": LightGBM Model

We trained a LightGBM model to act as an expert judge. It analyzes each of the ~500 candidates and assigns a precise relevance score.

Key Features (The Evidence)

The model's decisions are based on a rich set of features:

- **Semantic Similarity**: The cosine similarity between prefix and candidate embeddings.
- **Lexical Score**: The relevance score from our BM25 index.
- **Popularity Metrics**: Historical performance data like `orders`, `clicks`, and `volume` from the provided datasets.
- **String Features**: Simple but effective metrics like length ratios and character overlap.

Final Approach: Multi-Strategy, Prefix-First Retrieval

Core Idea

Prioritize reliable prefix candidates, then expand with fuzzy/lexical fallbacks.

Retrieval strategies (in order):

1. **Historical matches** (highest weight): learn which queries users actually searched for given a prefix historically.
2. **Direct prefix index**: fast lookup for queries starting with prefix (trie-like behavior).
3. **Shorter-prefix expansion**: try shorter prefixes to improve recall.
4. **Fuzzy matches**: lightweight fuzzy matching (e.g., `rapidfuzz`) for typos.
5. **Popular queries fallback**: global popular queries if above yield few results.

Results Summary

- **Runtime: 65 minutes** (within 90 min limit)
- **Throughput: 134 prefixes/second**
- **Memory:** Stable (no crashes)
- **Output:** `submission_fast.csv` (1.5GB)
- **Speedup from baseline: 531x** (24 days $\rightarrow$ 65 minutes)