



Supercharging Personalised Notifications At JobHai With ClickHouse

About Us



Sumit

Tech Lead at Infoedge

LinkdIn : <https://www.linkedin.com/in/sumit-kumar-121a2695/>

JOB ✓
hai



Arvind

Tech Lead at Infoedge

LinkdIn : <https://www.linkedin.com/in/arvind-saini-335556124/>

Agenda

What we'll cover today:

- 1 The Problem Statement
- 2 The Architecture & The Wall We Hit
- 3 The Insight & The Solution
- 4 Recommendation Engine Workflow
- 5 Technology Comparison: 5 Approaches
- 6 Why ClickHouse: Optimization & Cost Savings
- 7 Final Results: The Impact



The Problem Statement

The Goal: Send Personalized 'Best Jobs' to our Users.

The Scale:

- Users: Our active users within Last 1 month
- Inventory: 200K to 300K Qualifying Jobs
- Total Notifications: 10-20 Million

The Constraint:

- All notifications must be generated in < 60 Minutes (Earlier it was taking 8-9 hours)

The 'Anti-Skew' Rule:

- To prevent Job popularity bias, no single job can be recommended more than N times per day



The Architecture & The Wall We Hit



The Architecture:

- We treated this as a standard row-based transactional problem
- We iterated through users and found matching jobs using SQL joins

The Math (The Wall we hit):

- Single Thread Performance: Processing All the users sequentially took ~300,000 seconds (83 hours)
- With MySQL, we were running 10 parallel threads

The 'Brute Force' Fix:

- To meet the 1-hour SLA, we would have to run ~100 Parallel Threads on MySQL

The Limit:

- Scaling to 2x user base would require 200+ threads, which would increase our cost.

The Insight & The Solution



The Insight:

- We realized this isn't a "Transactional" problem (finding one record)
- It's an "Analytical" problem (filtering massive arrays)
- We moved to ClickHouse to utilize Vectorized Execution

The Solution:



Data Layout

Users & Jobs: Loaded into Memory tables for instant access

The Magic Query

```
SELECT user_id, job_id, score (Scoring Logic)
FROM users
CROSS JOIN jobs
WHERE
--- Filtering Logic ---
ORDER BY score
```

Dos and Don'ts of CROSS JOIN



✗ DON'T: Join All Data at Once

Problem:

If we join all users with all jobs at once, we hit memory constraints

Impact:

System crashes or extreme slowdown due to massive intermediate result sets

Why it fails:

20M users × 200K jobs = 4 Trillion potential combinations in memory

⚠ The Segmentation Trap

Problem:

Creating too many segments (e.g., by city) leads to too many queries

Impact:

Total execution time increases again due to query overhead and coordination

The Balance:

Need to find the sweet spot between segment size and query count

✓ DO: Optimize Segment Strategy

Solution:

Create segments based on strategic attributes (e.g., city/location)

Key:

Balance segment size to fit in memory while minimizing query count

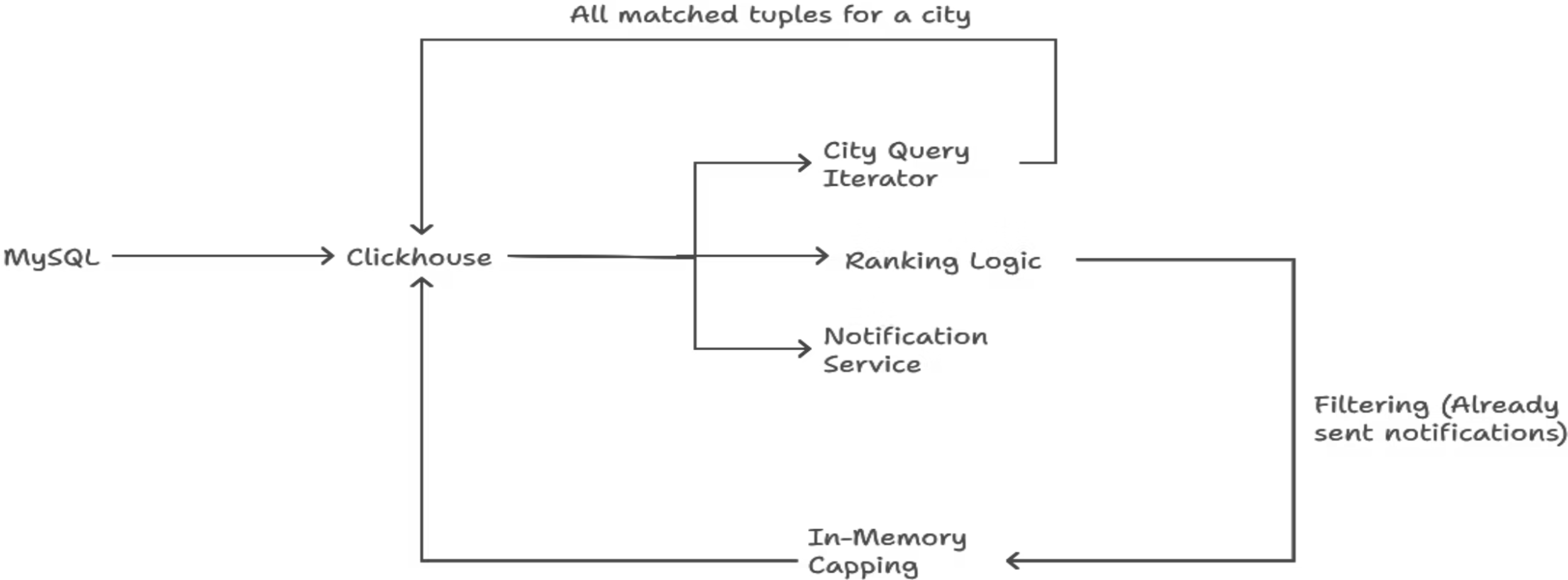
Result:

Optimal throughput with manageable memory footprint

Recommendation Engine Workflow



Data Flow for Job Ranking and Notification



Technology Comparison: 5 Approaches



Feature	MySQL (Old Approach)	ClickHouse (Our Choice)	Elasticsearch	Apache Spark	Apache Flink
Pattern	OLTP (Transactional)	OLAP (Analytical)	Search Engine	Distributed Batch	Stream Processing
Strategy	Brute Force Parallelism (100 Threads)	Vectorized Execution (Single Query)	Candidate Retrieval (User = Query)	Broadcast Join (In-Memory Cluster)	Continuous Event Loop (Real-time)
Throughput (1h SLA)	Failed without massive concurrency (50 mins @ 100 threads).	Excels. Hits SLA with minimal threads (< 10 mins).	High. Very fast, but high latency if "Deep Paging" occurs.	High. Built for large bursts, but has startup/shuffle overhead.	N/A. Spreads load over 24h (no "burst" required).
Est. Hardware (6M Users)	~64 vCPUs / 256 GB RAM (3-4 Large Read Replicas)	8 vCPUs / 64 GB RAM	~400 vCPUs / 768 GB RAM (12-15 Compute Nodes)	~200 vCPUs / 1.6 TB RAM (25-30 Executors)	~24 vCPUs / 64 GB RAM (4-6 Task Managers)
Primary Bottleneck	Connection Limits & Row Locking.	Memory Bandwidth (during large scans).	JVM Heap & Indexing overhead.	Data Shuffle & Serialization cost.	State Management (Global 5k Cap).
Cost Efficiency	Medium (High CPU usage for Joins).	Best (Lowest CPU/RAM footprint).	Worst (Requires massive RAM for Caches).	Medium (Large ephemeral cluster needed).	Good (Small footprint, but always-on).
Best For...	Single-record lookups (e.g., User Login).	High-speed filtering & Aggregations.	Fuzzy Matching (e.g., "React" ≈ "ReactJS").	Complex ETL & transformations.	Real-time triggers (e.g., "Job just posted").

Final Results: The Impact



< 30 mins

Time Saved

96% Faster Processing

\$0

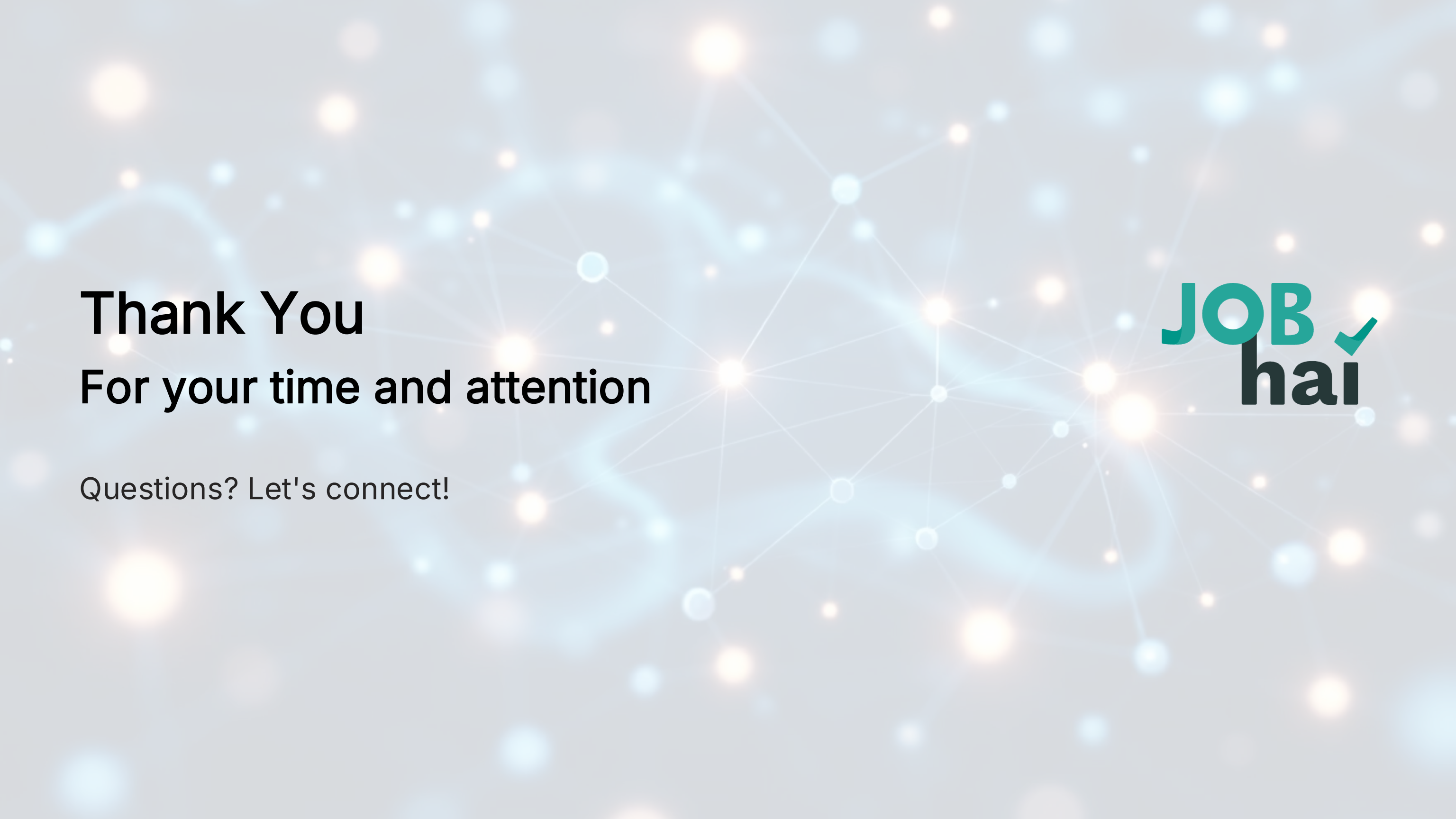
Cost Impact

NO Additional Cost

64GB RAM

Infrastructure (r5.2xlarge EC2)

Minimal Hardware Footprint



Thank You
For your time and attention

Questions? Let's connect!

JOB ✓
hai