# Enhancing SQL Server Searches with Elasticsearch and Python

As a seasoned SQL developer and tech enthusiast, I often encounter scenarios where traditional SQL Server searches struggle to meet performance and flexibility requirements. One such challenge is efficiently searching a large person database with multiple parameters and fuzzy logic. In this blog post, I'll explore how integrating Elasticsearch with SQL Server using Python can offer a robust solution to this problem.

This blog post is also available as a [notebook](https://github.com/Utsavv/PersonSearchUsingElastic/blob/main/Enhancing_SQL_Server_Searches_with_Elasticsearch_and_Python.ipynb)

## Business Problem

Imagine you're tasked with developing a search functionality for a vast person database. Users need the ability to search using various parameters such as:

* First Name
* Last Name
* Preferred Name
* City
* State
* ZipCode
* DOB
* Email

The complexity increases as searches can occur on any combination of these fields, and there's a need to implement fuzzy logic to handle typos and partial matches.

## Technical Challenges with SQL Server

### Inefficient Indexing

Creating indexes on every possible combination of the searchable columns isn't practical. The number of required indexes grows exponentially with each additional column, leading to:

* Increased storage requirements
* Slower write operations due to index maintenance
* Diminishing returns on query performance

### Implementing Fuzzy Logic

While SQL Server offers some capabilities like the LIKE operator or full-text search, these methods are limited:

* **Performance Issues**: Fuzzy searches can be slow on large datasets.
* **Complexity**: Implementing advanced fuzzy matching (e.g., handling typos, synonyms) requires intricate queries and functions that can degrade performance.

## High-Level Solution: Integrating Elasticsearch Using Python

To overcome these challenges, I recommend integrating Elasticsearch into your search architecture. Here's why:

* **Optimized for Search**: Elasticsearch is designed for lightning-fast full-text searches across multiple fields.
* **Fuzzy Matching**: It natively supports fuzzy logic, allowing for more flexible and user-friendly search experiences.
* **Scalability**: Built to handle large volumes of data and queries efficiently.
* **Existing Infrastructure**: Many production environments already use Elasticsearch for logging, making integration smoother.
* **Cost-Effective**: The basic version is free and sufficient for many use cases. For advanced security features, consider the paid tiers.

## Implementation Overview with Python

While Elasticsearch can be integrated using various programming languages, I'll demonstrate how to use Python for this purpose. Python's rich ecosystem and simplicity make it an excellent choice for interfacing with Elasticsearch.

### 1. Preparing Source Data

* Create a Sample Database: Use SQL Server to set up your person database, including all relevant fields.
* Populate Data: Insert sample records in sql server to simulate a realistic dataset.

%pip install pyodbc
%pip install elasticsearch

import os
import pyodbc
import random
import datetime
from elasticsearch import Elasticsearch, helpers
import time
from colorama import Fore, Style

# Define server name as a global variable so it can be set once
server = 'localhost'
db\_name = 'PersonSearchDB'

def create\_database\_if\_not\_exists(db\_name):
 # Connection to 'master' for creating the database
 master\_conn\_str = f'DRIVER={{ODBC Driver 17 for SQL Server}};SERVER={server};DATABASE=master;Trusted\_Connection=yes;'
 master\_conn = pyodbc.connect(master\_conn\_str)

 # Set autocommit to True to disable transactions
 master\_conn.autocommit = True

 master\_cursor = master\_conn.cursor()

 # Check if the database exists, if not, create it
 create\_db\_query = f"IF DB\_ID('{db\_name}') IS NULL CREATE DATABASE {db\_name};"
 master\_cursor.execute(create\_db\_query)

 # Close master connection
 master\_cursor.close()
 master\_conn.close()

# Function to return SQL Server connection
def get\_sql\_connection(db\_name):
 """
 Returns a SQL Server connection object for the given database name.
 If the database does not exist, the function connects to 'master' first to create it.

 Parameters:
 db\_name (str): The name of the database to connect to.

 Returns:
 pyodbc.Connection: A connection object to the specified database.
 """

 # Connection to the newly created or existing database
 conn\_str = f'DRIVER={{ODBC Driver 17 for SQL Server}};SERVER={server};DATABASE={db\_name};Trusted\_Connection=yes;'
 return pyodbc.connect(conn\_str)

def execute\_SQL\_Query(db\_name, query, params=None):
 """
 Executes a SQL query on the given database. Uses the connection obtained from get\_sql\_connection.

 Parameters:
 db\_name (str): The name of the database.
 query (str): The SQL query to execute.
 params (tuple): Parameters to pass to the query (optional).

 Returns:
 list: The result of the query if it's a SELECT query, otherwise None.
 """
 conn = get\_sql\_connection(db\_name)
 cursor = conn.cursor()
 if params:
 cursor.execute(query, params)
 else:
 cursor.execute(query)

 if query.strip().upper().startswith("SELECT"):
 result = cursor.fetchall() # Fetch all results if it's a SELECT query
 else:
 conn.commit()
 result = None

 cursor.close()
 conn.close()

 return result

def setup\_database\_and\_bulk\_insert\_data(db\_name, record\_count=1000, batch\_size=100):
 # List of Indian first names, last names, cities, and states
 first\_names = [
 'Rahul', 'Anjali', 'Amit', 'Pooja', 'Rajesh', 'Sneha', 'Vikram', 'Neha', 'Suresh', 'Sunita',
 'Arjun', 'Kiran', 'Ravi', 'Priya', 'Nikhil', 'Meera', 'Kunal', 'Rina', 'Aakash', 'Divya',
 'Sanjay', 'Anita', 'Deepak', 'Kavita', 'Manish', 'Shweta', 'Rohit', 'Preeti', 'Vijay', 'Swati',
 'Ajay', 'Nisha', 'Gaurav', 'Shalini', 'Alok', 'Tanvi', 'Varun', 'Shruti', 'Vivek', 'Rashmi'
 ]
 last\_names = [
 'Sharma', 'Patel', 'Gupta', 'Mehta', 'Jain', 'Agarwal', 'Reddy', 'Singh', 'Kumar', 'Verma',
 'Chopra', 'Desai', 'Iyer', 'Joshi', 'Kapoor', 'Malhotra', 'Nair', 'Pandey', 'Rao', 'Saxena',
 'Bose', 'Chatterjee', 'Das', 'Mukherjee', 'Banerjee', 'Bhat', 'Pillai', 'Menon', 'Choudhury', 'Trivedi',
 'Shah', 'Parekh', 'Chauhan', 'Patil', 'Dutta', 'Nayar', 'Kulkarni', 'Bhattacharya', 'Hegde', 'Sinha'
 ]
 cities = [
 'Mumbai', 'Delhi', 'Bangalore', 'Chennai', 'Hyderabad', 'Ahmedabad', 'Kolkata', 'Pune', 'Jaipur', 'Lucknow',
 'Surat', 'Kanpur', 'Nagpur', 'Indore', 'Thane', 'Bhopal', 'Visakhapatnam', 'Patna', 'Vadodara', 'Ghaziabad'
 ]
 states = [
 'MH', 'DL', 'KA', 'TN', 'TS', 'GJ', 'WB', 'MH', 'RJ', 'UP',
 'MP', 'AP', 'BR', 'HR', 'PB', 'KL', 'OR', 'AS', 'JK', 'CH'
 ]

 def random\_name():
 return random.choice(first\_names), random.choice(last\_names)

 def random\_email(first\_name, last\_name):
 return f"{first\_name.lower()}.{last\_name.lower()}@randommail.com"

 def random\_DOB():
 start\_date = datetime.date(1950, 1, 1)
 end\_date = datetime.date(2005, 12, 31)
 time\_between\_dates = end\_date - start\_date
 days\_between\_dates = time\_between\_dates.days
 random\_number\_of\_days = random.randrange(days\_between\_dates)
 random\_date = start\_date + datetime.timedelta(days=random\_number\_of\_days)
 return random\_date

 def random\_zipcode():
 return str(random.randint(100000, 999999)) # Indian zip codes are 6 digits

 # Create Persons table using execute\_SQL\_Query
 create\_table\_query = '''
 IF OBJECT\_ID('Persons', 'U') IS NOT NULL DROP TABLE Persons;
 CREATE TABLE Persons (
 FirstName NVARCHAR(50),
 LastName NVARCHAR(50),
 PreferredName NVARCHAR(50),
 City NVARCHAR(50),
 State NVARCHAR(50),
 ZipCode NVARCHAR(10),
 DOB DATE,
 Email NVARCHAR(100)
 );
 '''
 execute\_SQL\_Query(db\_name, create\_table\_query)

 # Insert records in batches
 for batch\_start in range(0, record\_count, batch\_size):
 values = []
 for \_ in range(batch\_size):
 first\_name, last\_name = random\_name()
 preferred\_name = first\_name # Assume preferred name is the first name
 city = random.choice(cities)
 state = random.choice(states)
 zipcode = random\_zipcode()
 dob = random\_DOB()
 dob\_str = dob.strftime('%Y-%m-%d')
 email = random\_email(first\_name, last\_name)
 values.append(f"SELECT '{first\_name}', '{last\_name}', '{preferred\_name}', '{city}', '{state}', '{zipcode}', '{dob\_str}', '{email}'")

 # Create bulk insert query using INSERT INTO ... SELECT
 insert\_query = '''
 INSERT INTO Persons (FirstName, LastName, PreferredName, City, State, ZipCode, DOB, Email)
 ''' + " UNION ALL ".join(values)

 execute\_SQL\_Query(db\_name, insert\_query)
 print(f"Inserted batch starting at record {batch\_start}")

 #Master Database Creation
create\_database\_if\_not\_exists(db\_name)

#Setup Database and Bulk Insert Data
#I am inserting million records to demo perfromance gains. Feel free to adjust for testing purpose.
setup\_database\_and\_bulk\_insert\_data(db\_name, record\_count=1000000, batch\_size=10000)

### 2. Setting Up Elasticsearch

* Download and Install: Obtain Elasticsearch from the official website and follow the installation guide.
* Configuration: Adjust settings as needed, such as cluster name and network configurations.

### 3. Creating and Populating Elasticsearch Index

Define Mappings: Specify how each field should be indexed and analyzed, particularly those requiring fuzzy search capabilities.

#### Data Extraction and Bulk Insertion

Extract data from SQL Server and load it into Elasticsearch.

from elasticsearch import Elasticsearch, helpers
import urllib3

# Suppress warnings about insecure connections (optional)
urllib3.disable\_warnings(urllib3.exceptions.InsecureRequestWarning)

# Replace with your actual password
elastic\_password = "IMoWOv8DHTMNnQod37NS"

# Initialize the Elasticsearch client with SSL and authentication
# make sure elastic search is running on port 9200,
# Use the following command to start elastic search
# .\elasticsearch-8.11.1\bin\elasticsearch.bat

es = Elasticsearch(
 ["https://localhost:9200"],
 ca\_certs=False, # Disable SSL certificate verification
 verify\_certs=False, # Disable SSL cert verification (use with caution)
 basic\_auth=("elastic", elastic\_password),
)

def index\_data\_to\_elasticsearch(db\_name, index\_name, batch\_size=10000):

 # Delete the existing index if it exists
 if es.indices.exists(index=index\_name):
 es.indices.delete(index=index\_name)
 print(f"Deleted existing index: {index\_name}")

 # Create a new index with mappings
 index\_mappings = {
 "mappings": {
 "properties": {
 "FirstName": {"type": "text"},
 "LastName": {"type": "text"},
 "PreferredName": {"type": "text"},
 "City": {"type": "text"},
 "State": {"type": "keyword"},
 "ZipCode": {"type": "keyword"},
 "DOB": {"type": "date"},
 "Email": {"type": "keyword"}
 }
 }
 }
 es.indices.create(index=index\_name, body=index\_mappings)
 print(f"Created new index: {index\_name}")

 # Get total number of records
 total\_records\_query = "SELECT COUNT(\*) FROM Persons"
 total\_records\_result = execute\_SQL\_Query(db\_name, total\_records\_query)
 total\_records = total\_records\_result[0][0]

 offset = 0
 while offset < total\_records:
 query = f'''
 SELECT FirstName, LastName, PreferredName, City, State, ZipCode, DOB, Email
 FROM Persons
 ORDER BY FirstName
 OFFSET {offset} ROWS FETCH NEXT {batch\_size} ROWS ONLY
 '''
 rows = execute\_SQL\_Query(db\_name, query)
 actions = []
 for row in rows:
 print(f"\n\nrow: {row}")
 doc = {
 "\_index": index\_name,
 "\_source": {
 "FirstName": row[0],
 "LastName": row[1],
 "PreferredName": row[2],
 "City": row[3],
 "State": row[4],
 "ZipCode": row[5],
 "DOB": row[6].strftime('%Y-%m-%d') if row[6] else None,
 "Email": row[7]
 }
 }
 actions.append(doc)

 try:
 helpers.bulk(es, actions)
 except helpers.BulkIndexError as e:
 print(f"Bulk indexing error: {e}")
 for error in e.errors:
 print(error)

 offset += batch\_size
 print(f"Indexed {offset}/{total\_records} records")

# Example usage: setting up the database and bulk inserting data
# setup\_database\_and\_bulk\_insert\_data(db\_name, record\_count=1000, batch\_size=100)
index\_data\_to\_elasticsearch('PersonSearchDB', 'person\_index', batch\_size=1000)

### 4. Implementing Search Queries

* Basic Searches: Perform searches on single or multiple fields.
* Fuzzy Searches: Utilize Elasticsearch's fuzzy query capabilities to handle misspellings and partial matches.
* Boolean search:

def ExecuteElasticSearch(search\_query):
 batch\_size=10
 # Execute the search
 response = es.search(index="people\_index", body=search\_query, size=batch\_size)

 # Process the results
 print(f"Found {response['hits']['total']['value']} documents:")
 for hit in response['hits']['hits']:
 print(f"\n\n\n{hit['\_source']}")

def first\_name\_search(column\_name, first\_name):
 search\_query = {
 "query": {
 "match": {
 column\_name: first\_name
 }
 }
 }
 ExecuteElasticSearch(search\_query)

def multi\_field\_wildcard\_search(first\_name, last\_name, birth\_year):
 search\_query = {
 "query": {
 "bool": {
 "must": [
 {"wildcard": {"FirstName": f"\*{first\_name}\*"}},
 {"wildcard": {"LastName": f"\*{last\_name}\*"}},
 {
 "range": {
 "BirthDate": {
 "gte": f"{birth\_year}-01-01",
 "lte": f"{birth\_year}-12-31"
 }
 }
 }
 ]
 }
 }
 }
 ExecuteElasticSearch(search\_query)

def fuzzy\_logic\_search(first\_name):
 search\_query = {
 "query": {
 "fuzzy": {
 "FirstName": {
 "value": first\_name,
 "fuzziness": 2
 }
 }
 }
 }
 ExecuteElasticSearch(search\_query)

def boolean\_logic\_search(first\_name, last\_name, birth\_year):
 search\_query = {
 "query": {
 "bool": {
 "should": [
 {"wildcard": {"FirstName": f"\*{first\_name}\*"}},
 {"wildcard": {"LastName": f"\*{last\_name}\*"}},
 {
 "range": {
 "BirthDate": {
 "gte": f"{birth\_year}-01-01",
 "lte": f"{birth\_year}-12-31"
 }
 }
 }
 ],
 "minimum\_should\_match": 1
 }
 }
 }
 ExecuteElasticSearch(search\_query)

# Example usage of search functions
first\_name\_search("FirstName", "Rahul")

#Multi Field Wildcard Search
multi\_field\_wildcard\_search("Rahul", "Sharma", 1990)

#Fuzzy Logic Search
fuzzy\_logic\_search("Rahul")

#Boolean Logic Search
boolean\_logic\_search("Rahul", "Sharma", 1990)

### 5. Performance Comparison

To illustrate the benefits:

* **Unindexed SQL Server Searches**: Execute searches without indexes to highlight performance issues.
* **Elasticsearch Searches**: Run equivalent searches and compare response times and accuracy.

def compare\_performance(first\_name, last\_name, preferred\_name, iterations=1000):
 # SQL query
 sql\_query = '''
 SELECT FirstName, LastName, PreferredName
 FROM Persons
 WHERE FirstName = ? AND LastName = ? AND PreferredName LIKE ?
 '''

 # Warm-up run for SQL Server
 execute\_SQL\_Query(db\_name, sql\_query, (first\_name, last\_name, preferred\_name))

 # Record execution times for SQL Server
 sql\_execution\_times = []
 for \_ in range(iterations):
 start\_time = time.perf\_counter()
 execute\_SQL\_Query(db\_name, sql\_query, (first\_name, last\_name, preferred\_name))
 end\_time = time.perf\_counter()
 sql\_execution\_times.append(end\_time - start\_time)

 # Calculate average execution time for SQL Server
 average\_time\_sql = sum(sql\_execution\_times) / iterations

 # Elasticsearch query
 es\_query = {
 "query": {
 "bool": {
 "must": [
 {"match": {"FirstName": first\_name}},
 {"match": {"LastName": last\_name}},
 {"wildcard": {"PreferredName": preferred\_name}}
 ]
 }
 }
 }

 # Warm-up run for Elasticsearch
 es.search(index="person\_index", body=es\_query)

 # Record execution times for Elasticsearch
 es\_execution\_times = []
 for \_ in range(iterations):
 start\_time = time.perf\_counter()
 es.search(index="person\_index", body=es\_query)
 end\_time = time.perf\_counter()
 es\_execution\_times.append(end\_time - start\_time)

 # Calculate average execution time for Elasticsearch
 average\_time\_es = sum(es\_execution\_times) / iterations

 print(f"\033[91mAverage SQL execution time: {average\_time\_sql:.6f} seconds\033[0m")
 print(f"\033[92mAverage Elasticsearch execution time: {average\_time\_es:.6f} seconds\033[0m")
 performance\_gain = (average\_time\_sql - average\_time\_es) / average\_time\_sql \* 100
 print(f"\033[93mElasticsearch performance gain: {performance\_gain:.2f}%\033[0m")

#Performance Comparison
compare\_performance("Rahul", "Sharma", "Rahul%")

On my system, I observed a performance improvement of approximately 100% when processing a million entries. With larger datasets, the performance gains are likely to increase further.

# Addressing Production Concerns

## Why Not Implement Python Code in SQL Server?

While SQL Server supports Python via Machine Learning Services, enabling this feature in production can pose security risks.

* **Security Risk**: Activating external scripting (sp\_configure 'external scripts enabled', 1) can expose your server to potential vulnerabilities.
* **Resource Overhead**: Running Python scripts within SQL Server can consume significant resources, affecting database performance.

## Hardware Requirements for Elasticsearch

* **Modest Initial Setup**: Elasticsearch can run on standard hardware for small to medium datasets.
* **Scalability**: For larger datasets and higher query volumes, plan for additional resources or consider a cluster setup.
* **Monitoring**: Keep an eye on CPU, memory, and disk I/O to ensure optimal performance.

# Conclusion

Integrating Elasticsearch with SQL Server can significantly enhance search capabilities, especially for large datasets requiring flexible and fuzzy searches. While it introduces additional components to your architecture, the performance gains and improved user experience can be well worth the effort.

Remember, handling PII necessitates a careful approach to security. By following best practices and possibly opting for Elasticsearch's paid tiers for advanced security features, you can create a powerful, secure, and efficient search solution.