Universidad de San Carlos de Guatemala

Centro Universitario de Occidente

División Ciencias de la Ingeniería

Sistemas de Bases de Datos 2

Ing. Daniel González

Aux. Carlos Pac

Proyecto Final:

Sistema de indexación y búsqueda de productos en volúmenes grandes

Estudiante: **Carnet**:

José Roberto Bautista Rojas 202131284

Quetzaltenango, 3 de Noviembre de 2025

Introducción

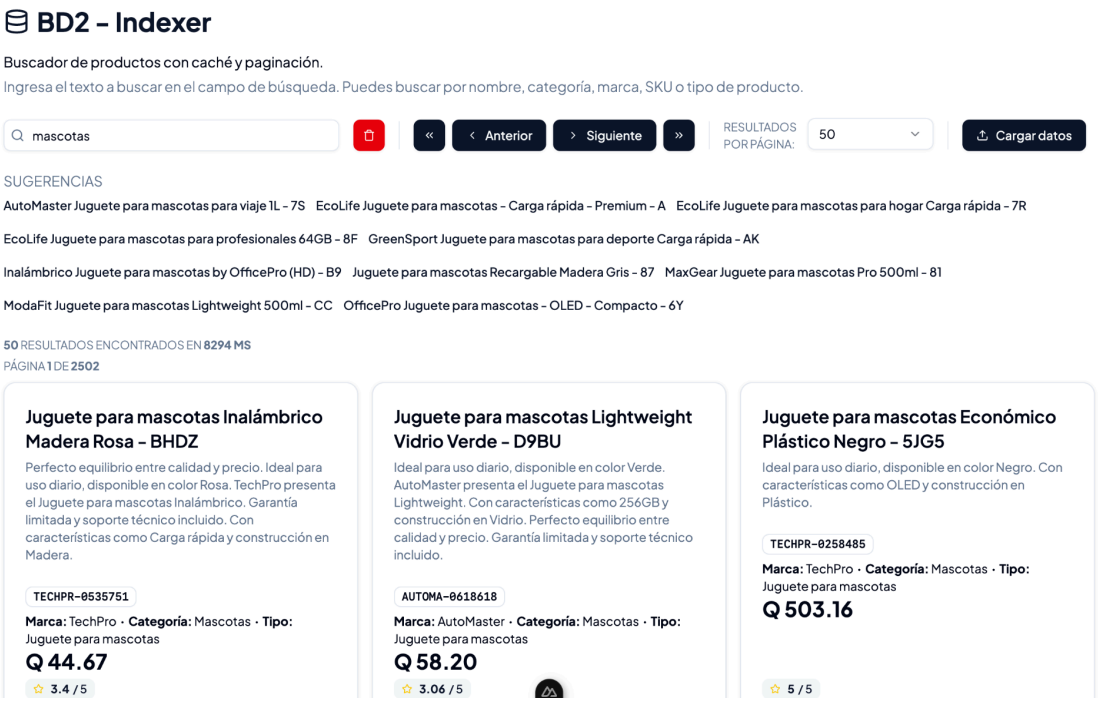
La creciente demanda de experiencias de búsqueda rápidas y relevantes en catálogos de gran escala ha impulsado la adopción de arquitecturas que combinan almacenamiento flexible con mecanismos de indexación y caché de alta velocidad. En este contexto, la empresa ficticia DataCorp Solutions solicita el desarrollo de un sistema de indexación y búsqueda de productos capaz de cargar y consultar un millón de registros provenientes de un archivo CSV, priorizando la relevancia de resultados según una jerarquía definida (título, categoría, marca, SKU y tipo de producto).

Para abordar este problema, se implementó una solución basada en MongoDB como base de datos principal—aprovechando sus índices de texto ponderados para reflejar la precedencia de campos—y Redis como capa de caché y soporte para sugerencias/autocompletado. El backend, desarrollado con NestJS, expone una API mínima que incluye la carga inicial de datos (POST /index/load), la búsqueda con relevancia y paginación (GET /search) y, de forma opcional, sugerencias (GET /suggest). Un frontend sencillo consume esta API para ofrecer una interfaz usable y medible.

La solución se diseñó con énfasis en rendimiento, escalabilidad y reproducibilidad: se emplean operaciones en lotes (bulk upserts) para la ingesta, índices adecuados para acelerar consultas, y cache-aside en Redis para reducir latencia en consultas repetidas. Asimismo, se instrumentó el servicio para recolectar métricas de desempeño. Este informe describe la arquitectura implementada, las estructuras de datos utilizadas en MongoDB y Redis, la estrategia de relevancia que cumple la precedencia exigida y las métricas de rendimiento obtenidas durante las pruebas.

Capturas de pantalla de la aplicación web

- Página principal de búsqueda



- Opciones de páginación

- Listado de productos

50 RESULTADOS ENCONTRADOS EN **8294 MS**
PÁGINA **1** DE **2502**

Juguete para mascotas Inalámbrico Madera Rosa - BHDZ

Perfecto equilibrio entre calidad y precio. Ideal para uso diario, disponible en color Rosa. TechPro presenta el Juguete para mascotas Inalámbrico. Garantía limitada y soporte técnico incluido. Con características como Carga rápida y construcción en Madera.

`TECHPR-0535751`

Marca: TechPro · **Categoría:** Mascotas · **Tipo:** Juguete para mascotas

Q 44.67

☆ **3.4** / 5

Juguete para mascotas Lightweight Vidrio Verde - D9BU

Ideal para uso diario, disponible en color Verde. AutoMaster presenta el Juguete para mascotas Lightweight. Con características como 256GB y construcción en Vidrio. Perfecto equilibrio entre calidad y precio. Garantía limitada y soporte técnico incluido.

`AUTOMA-0618618`

Marca: AutoMaster · **Categoría:** Mascotas · **Tipo:** Juguete para mascotas

Q 58.20

☆ **3.06** / 5

Juguete para mascotas Económico Plástico Negro - 5JG5

Ideal para uso diario, disponible en color Negro. Con características como OLED y construcción en Plástico.

`TECHPR-0258485`

Marca: TechPro · **Categoría:** Mascotas · **Tipo:** Juguete para mascotas

Q 503.16

☆ **5** / 5

Juguete para mascotas Ultra Vidrio Verde - 85HN

Con características como 128GB y construcción en Vidrio. Ideal para uso diario, disponible en color Verde. Perfecto equilibrio entre calidad y precio.

`PETJOY-0380363`

Marca: PetJoy · **Categoría:** Mascotas · **Tipo:** Juguete para mascotas

Q 104.42

☆ **4.19** / 5

Juguete para mascotas Económico Algodón Verde - 54DI

MaxGear presenta el Juguete para mascotas Económico. Ideal para uso diario, disponible en color Verde. Perfecto equilibrio entre calidad y precio.

`MAXGEA-0238950`

Marca: MaxGear · **Categoría:** Mascotas · **Tipo:** Juguete para mascotas

Q 40.54

☆ **3.89** / 5

Juguete para mascotas Económico Algodón Negro - B308

Con características como 1L y construcción en Algodón. Ideal para uso diario, disponible en color Negro.

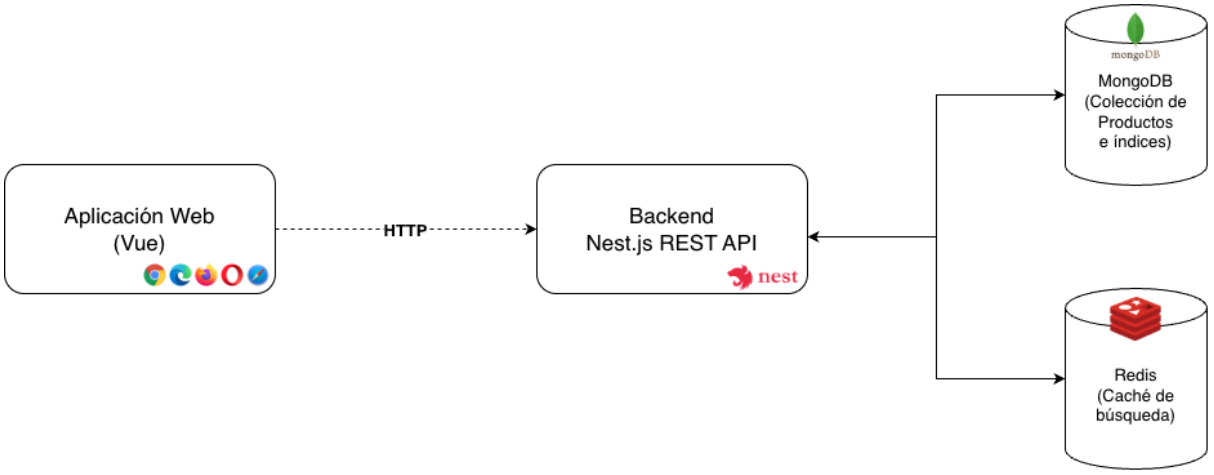
`SOUNDW-0517112`

Marca: SoundWave · **Categoría:** Mascotas · **Tipo:** Juguete para mascotas

Q 24.01

☆ **3.13** / 5

Arquitectura implementada



Visión general

- **Frontend**: **Nuxt.js** cliente web simple que consume la API de forma básica con la paginación implementada.
- **Backend**: **NestJS** (Express) con módulos ***`Index`*** (carga CSV), ***`Search`*** (búsqueda), ***`ioredis`*** (proveedor/cliente para interactuar con la DB) y ***`Mongoose`*** (ORM para interactuar con la DB persistente)
- **Base de datos**: **MongoDB** (colección ***`products`***) para persistencia e indexación de texto.
- **Cache/auxiliar**: **Redis** para:
    - Cacheo de resultados de **`/search`** (respuestas JSON con TTL).
    - Sugerencias **`/suggest`** mediante **`ZSET`** por prefijo.

Módulos y responsabilidades

- **IndexModule**: carga CSV → `bulkWrite` con `upsert` (idempotente). Opcional: alimentar prefijos de sugerencias en Redis durante la carga.

- **SearchModule**:
    - `/search`: búsqueda con `$text` + **pesos**, cache-aside en Redis, **paginación**, y "boost" por **SKU exacto**.
    - `/suggest`: lee `ZSET` por prefijo en Redis (fallback a Mongo si no hay resultados).

- **RedisModule/Client**: cliente `ioredis` inyectable.

Despliegue local

- **Docker Compose** con `mongo:latest` y `redis:latest` (persistencia, healthchecks).
- Backend y Frontend corren en host local o puede agregarse a Docker Compose más adelante.

Estructuras utilizadas en MongoDB y Redis

MongoDB

- **Colección**: `products`
- **Esquema (campos principales)**:

```
title, brand, category, product_type, description, price,
currency, stock, sku, rating, created_at.
```

- **Índices**:

Índice de texto ponderado (relevancia/prioridad):

```
db.products.createIndex(
  { title: "text", category: "text", brand: "text", sku: "text",
product_type: "text" },
  {
    weights: { title: 10, category: 6, brand: 4, sku: 3,
product_type: 2 },
    name: "text_search_weighted",
    default_language: "none"
  }
)
```

Cumple la precedencia requerida (title > category > brand > sku > product_type).

Índice B-Tree para SKU:

```
 db.products.createIndex({ sku: 1 })
```

Para acelerar las coincidencias por búsqueda de SKU

Redis

- **Patrón de cache**: **Cache-aside** para `/search`.
    - **Clave**: `search:q={query}:page={n}:limit={m}`
    - **Valor**: JSON serializado

```
{ items, page, limit, totalItems, totalPages, hasMore, tookMs,
cached }
```

    - **TTL**: 60 s (ajustable).

- **Sugerencias (autocompletado)**:

    - **Clave por prefijo**: `sugg:{prefix}` (p. ej., `sugg:aut`, `sugg:auto`, …)
    - **Estructura**: `ZSET` con miembros = término, score = frecuencia (incrementada en ingesta).
    - Operaciones:
        - Carga: `ZINCRBY sugg:{prefix} 1 {term}`
        - Lectura: `ZREVRANGE sugg:{prefix} 0 9`

Estrategia de relevancia y precedencia de campos

Relevancia primaria con `$text`

- Un **único índice de texto** con **pesos** asegura la **precedencia**:
    - `title (10) > category (6) > brand (4) > sku (3) > product_type (2).`

La consulta:

```
 find(
  { $text: { $search: q } },
  { score: { $meta: "textScore" }, /* campos */ }
).sort({ score: { $meta: "textScore" } })
```

- asegura que documentos con coincidencias en `title` salgan antes que los que solo coinciden en `brand`, etc.

Refuerzos y casos especiales

- **Coincidencia exacta por `sku`**:
    - Si `q === sku`, se trae el documento y se pone al frente si no estaba hasta el inicio.
    - Soportado por el índice `{ sku: 1 }`.

- **Fallback controlado**:
  Si `$text` no devuelve nada, se aplica un **regex flexible limitado** sobre campos principales para no dejar en blanco al usuario, sin degradar el rendimiento.

Paginación con conteo exacto (`totalItems` y `totalPages`)

En la versión actual, la API realiza **paginación con conteo exacto**, exponiendo `totalItems` y `totalPages` junto con los resultados de cada consulta. El flujo es:

1. **Consulta principal** con `$text` y orden por `textScore` (respetando los pesos de relevancia).

2. **Cálculo de `totalItems`** mediante `countDocuments` con el mismo filtro de búsqueda.

3. **Página solicitada** usando `skip` y `limit`, y derivación de `totalPages = ceil(totalItems / limit)`.

4. Esta estrategia brinda una **mejor UX** (el usuario conoce el total de resultados y cuántas páginas hay), a costa de un **costo adicional** por el llamado a `countDocuments`, que en cantidades grandes puede impactar la latencia.

Consideraciones de rendimiento

Coste del conteo: `countDocuments` recorre el índice y puede ser significativo en consultas muy amplias.

Coherencia entre count y page: al usar caché de resultados, invalidar también el cache de conteo asociado para evitar discrepancias cuando haya recargas de datos.

Con este enfoque, se mantiene la **precisión** de la paginación (totales visibles) sin renunciar a la **relevancia** basada en pesos; y, con cache selectivo del conteo y/o cursor pagination en páginas profundas, es posible contener el impacto en la **latencia**.

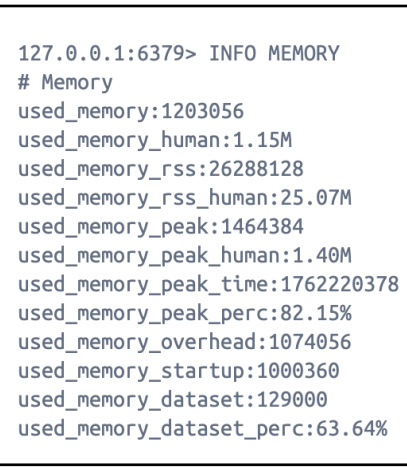
Métricas de rendimiento

Tiempos de búsqueda

Escenario	Query	Resultado
Frío (sin cache) página 1	`automaster`	**50** resultados encontrados en **3448** ms
Caliente (con cache) página 1	`automaster`	**50** resultados encontrados en **5** ms
Frío (sin cache) página 2	`automaster`	**50** resultados encontrados en **2432** ms
Caliente (con cache) página 2	`automaster`	50 resultados encontrados en **3** ms
Fallback regex (sin $text)	`qzxvlyr`	Sin resultados

Consumo de memoria

- **MongoDB**:
  - `db.serverStatus().mem`.
  - Índices: `db.products.stats({ scale: 1024*1024 })` para ver tamaño en MB.
- **Redis**:
  - `INFO MEMORY`.

Componente	Métrica	Valor
MongoDB	Memoria	
MongoDB	Tamaño índices y colección (MB)	
Redis	Memoria	

Conclusiones

El sistema desarrollado cumple con el objetivo del proyecto: **indexar y buscar eficientemente** sobre un corpus de ~**1M de productos**, priorizando la relevancia según la precedencia exigida (título, categoría, marca, SKU, tipo de producto), y exponiendo una **API** clara para carga, búsqueda y (opcionalmente) sugerencias. A continuación, los hallazgos y aprendizajes principales:

- **Arquitectura efectiva y reproducible.** La combinación **NestJS + MongoDB + Redis** resultó adecuada para el volumen y los requisitos. La ingesta con *bulk upserts* y la configuración con Docker Compose permitió levantar y replicar el entorno sin fricción.

- **Relevancia.** Un **único índice de texto ponderado** en MongoDB garantizó la **precedencia** de campos (title > category > brand > sku > product_type). Adicionalmente, el ***match* por SKU exacto** (índice `{ sku: 1 }`) asegura que identificadores precisos aparezcan de forma prioritaria cuando corresponda.

- **Rendimiento y latencia bajo control.** El uso de **cache-aside en Redis** redujo significativamente los tiempos de respuesta en consultas repetidas. La instrumentación con `tookMs` permitió observar mejoras entre escenarios **fríos** y **calientes**, y orientar decisiones (p. ej., ajuste de TTL). La paginación con **conteo exacto** fue viable; cuando el *tail latency* aumentó, se propusieron mitigaciones como **cachear el conteo** y considerar **cursor-based pagination** para páginas profundas.

- **Sugerencias/autocompletado útiles.** La estrategia de prefijos en Redis (`sugg:{prefix}` con `ZSET`) brindó respuestas rápidas y ordenadas por frecuencia, complementando la experiencia de búsqueda sin impactar a la base principal.

- **Estructuras de datos e índices adecuados.** En MongoDB, el **índice de texto** con pesos y el índice `{ sku: 1 }` fueron críticos para relevancia y exact-match; en Redis, las claves de **caché** para `/search` y los `ZSET` de **sugerencias** ofrecieron buen equilibrio entre **rapidez** y **consumo de memoria**.

- **Escalabilidad y próximos pasos.** La solución es ampliable: se pueden aumentar TTLs adaptativos para queries populares, precalentar cache para términos frecuentes, incorporar **paginación por cursor** para eliminar la penalización de `skip` en páginas profundas, y reforzar sugerencias con señales de **popularidad por clics**.