

UNIVERSIDADE FEDERAL DE VIÇOSA  
CAMPUS FLORESTAL

# Trabalho Prático 3

## Gestão, Recuperação e Análise de Informação

Jonathan Lopes - 2666  
Ricardo Spinola - 3471  
Guilherme Correa Souza - 3509  
Heron Fillipe - 4211

Trabalho Prático apresentado à disciplina de  
Gestão, Recuperação e Análise de Informação  
do curso de Ciência da Computação da Univer-  
sidade Federal de Viçosa.

Florestal  
Novembro de 2023

# CCF 424 - Gestão, Recuperação e Análise de Informação

## TP 03 - Sistema de Recuperação da Informação

Jonathan Lopes - 2666

Ricardo Spinola - 3471

Guilherme Correa Souza - 3509

Heron Fillipe - 4211

28 de Novembro de 2023

## Sumário

<b>1</b>	<b>Introdução</b>	<b>3</b>
<b>2</b>	<b>Desenvolvimento</b>	<b>3</b>
2.1	Arquitetura . . . . .	3
2.2	Definição do Contexto . . . . .	3
2.3	Definição do SGBD . . . . .	4
2.3.1	MongoDB como Sistema de RI . . . . .	4
2.4	Definição do Sistema de Recuperação de Informação . . . . .	5
2.5	Front-End - Interação com o Sistema . . . . .	5
<b>3</b>	<b>Sistema de RI</b>	<b>5</b>
3.1	Entendimento do Contexto e Relevância para os Usuários . . . . .	5
3.2	Modelo de Recuperação . . . . .	6
3.3	Pré-Processamento de Texto e Indexação . . . . .	6
3.3.1	Analisadores . . . . .	6
3.3.2	Tokenizadores . . . . .	6
3.3.3	Mapeando os atributos para indexação . . . . .	8
3.3.4	Indexação e Inserção em Massa (Bulk Insert) . . . . .	9
3.4	Recuperando Dados . . . . .	10
3.4.1	Rankeamento . . . . .	10
3.4.2	Filtragem por Nome . . . . .	11
3.4.3	Filtragem Avançada . . . . .	12
3.5	Inserindo Novo Documento . . . . .	13
3.6	Desabilitando recursos do RI . . . . .	14
3.7	Performance . . . . .	15
<b>4</b>	<b>Exemplos</b>	<b>16</b>
4.1	Pesquisa por nome . . . . .	16
4.2	Pesquisa avançada . . . . .	17
4.3	Inserção de novo documento . . . . .	18
4.4	Visualização e exclusão de documentos inseridos . . . . .	18



# 1 Introdução

Este relatório descreve as fases envolvidas no desenvolvimento de um sistema de recuperação de informações. Propusemos um modelo voltado para a recuperação de jogos, com foco nos jogos disponíveis na Steam. O banco de dados inclui 78.581 jogos, cada um com várias características que possibilitam uma filtragem complexa durante a recuperação dos dados.

Durante o desenvolvimento, testamos várias ferramentas para encontrar a mais adequada às nossas necessidades. Experimentamos aplicações como Redis, Sonic e Apache. Inicialmente, consideramos o MongoDB como o principal sistema para a recuperação de informações, mas logo identificamos limitações em seu conjunto de ferramentas para essa finalidade.

Consequentemente, optamos por uma stack composta por Docker, SolidJS(Front), NodeJS(Server), MongoDB(SGBD) e ElasticSearch(RI).

## 2 Desenvolvimento

### 2.1 Arquitetura

Na Figura 1 a seguir, apresentamos de forma simplificada a arquitetura deste sistema. Essa representação oferece uma visão clara dos componentes utilizados e de como eles interagem entre si.

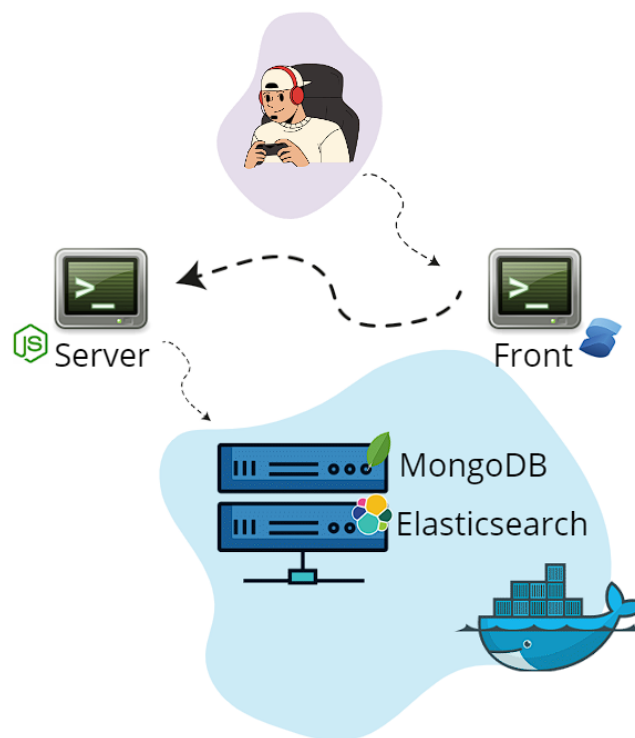


Figura 1: Arquitetura do sistema de RI.

### 2.2 Definição do Contexto

A coleção de documentos utilizada neste trabalho baseia-se em informações de jogos da Steam. Ao todo são 78.581 jogos, cada um com características específicas<sup>1</sup>, conforme ilustrado na Figura 2 abaixo:

---

<sup>1</sup>Realizamos uma limpeza e tratamento dos dados, removendo elementos desnecessários e adaptando os nomes dos campos

```

AppID: 570
name: "Dota 2"
release_date: "2013-07-09"
estimated_owners: "100000000 - 200000000"
required_age: 0
price: 0
description: "The most-played game on Steam. Every day, millions of players worldwid..."
languages: "Bulgarian, Czech, Danish, Dutch, English, Finnish, French, German, Gre..."
image: "https://cdn.akamai.steamstatic.com/steam/apps/570/header.jpg?t=1658774..."
windows: "True"
mac: "True"
linux: "True"
meta_score: 90
user_score: 0
positive: 1477153
negative: 300437
recommendations: 14300
developers: "Valve"
publishers: "Valve"
categories: "Multi-player,Co-op,Steam Trading Cards,Steam Workshop,SteamVR Collecti..."
genres: "Action,Free to Play,Strategy"
screenshots: "https://cdn.akamai.steamstatic.com/steam/apps/570/ss_ad8eee787704745cc..."

```

Figura 2: Características dos dados

## 2.3 Definição do SGBD

Neste parte do trabalho, percebemos que uma única ferramenta não seria suficientemente boa para cumprir dois propósitos: armazenar dados de forma consistente e eficiente, e realizar buscas rápidas e eficazes. Por isso, optamos pelo MongoDB(SGBD) para o armazenamento dos dados e para buscas por informações adicionais. Na seção sobre o Sistema de Recuperação de Informações, detalharemos melhor essa divisão entre os sistemas de armazenamento e recuperação (Elasticsearch).

### 2.3.1 MongoDB como Sistema de RI

No decorrer do desenvolvimento do projeto, nossa escolha inicial foi implementar índices no MongoDB para facilitar a recuperação de informações. Contudo, durante o processo, nos deparamos com limitações no MongoDB que nos levaram a considerar alternativas. Os aspectos em que o MongoDB se mostrou insuficiente incluem:

**Busca por texto limitada:** Apesar da possibilidade de criar índices no MongoDB, seu conjunto limitado de utilitários revelou-se um ponto negativo. Funcionalidades como relevância, ranqueamento e fuzzy (tolerância a erros) não estavam disponíveis.

**Flexibilidade na indexação e recuperação:** Os dados variam além de textos, e o MongoDB mostrou-se inflexível nesse aspecto. Sua indexação estava restrita ao uso de vários campos combinados em um único índice de texto, impossibilitando pesquisas individuais por campos. Outra limitação era a tokenização, que não podia ser personalizada para o nosso contexto. Detalharemos mais sobre esta questão em breve.



Figura 3: Index no MongoDB.

## 2.4 Definição do Sistema de Recuperação de Informação

Tendo em mente as funcionalidades necessárias, iniciamos a busca por uma ferramenta que atendesse aos nossos requisitos. Após um período testando soluções mais simples, decidimos pelo Elasticsearch.

Esta é uma ferramenta poderosa que oferece diversas funcionalidades para a construção de um sistema de recuperação de informações (RI) robusto. Nesta seção, não detalharemos suas características; essas serão exploradas ao compararmos os conceitos abordados neste trabalho com os apresentados na disciplina, na seção Sistema de RI.

## 2.5 Front-End - Interação com o Sistema

Como mencionado anteriormente, escolhemos o SolidJS para desenvolver uma interface interativa que permite visualizar as informações pesquisadas. Não abordaremos detalhes de sua implementação aqui, pois reconhecemos que não é o foco principal deste trabalho. A seguir, apresentamos um exemplo de layout:

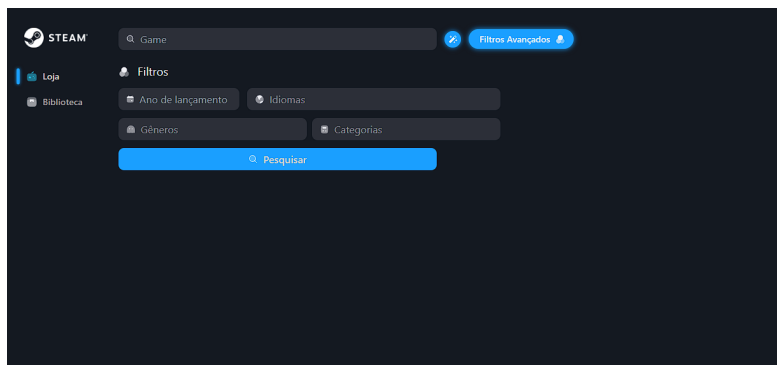


Figura 4: Interface de usuário.

# 3 Sistema de RI

Esta seção detalha os principais processos envolvidos na construção do sistema de recuperação de informações (RI), abordando os conceitos utilizados e suas aplicações.

## 3.1 Entendimento do Contexto e Relevância para os Usuários

Antes de desenvolver o sistema de RI, foi crucial compreender as necessidades dos usuários e as características dos dados. O sistema de RI foi projetado considerando a interação do usuário com o sistema e suas expectativas de resposta.

O foco principal é possibilitar que o usuário filtre jogos, priorizando a relevância com base em recomendações de jogadores e avaliações positivas. A busca pode ser ampliada para incluir características como categoria, gênero, data de lançamento e idioma.

## 3.2 Modelo de Recuperação

No Elasticsearch, adotamos dois modelos principais para a busca e recuperação de informações: o modelo de espaço vetorial e o modelo probabilístico. O modelo de espaço vetorial é usado para medir a similaridade entre consultas e documentos, ordenando os resultados por relevância. Por outro lado, o modelo probabilístico, empregando o algoritmo BM25, estima a probabilidade de um documento ser relevante para a consulta, levando em conta fatores como a frequência dos termos.

## 3.3 Pré-Processamento de Texto e Indexação

### 3.3.1 Analisadores

No Elasticsearch, os analisadores desempenham um papel no pré-processamento dos dados antes de sua tokenização e indexação. Este passo é essencial, pois os dados frequentemente não estão organizados de forma ideal para a criação de índices invertidos. A Figura 5 ilustra esse processo. Os filtros são utilizados para tratar os campos: por exemplo, *lowercase* converte todo o texto para minúsculas, *asciifolding* remove caracteres especiais e *trim* elimina espaços extras.

```
analyzer: {
  game_analyzer_complete: {
    type: 'custom',
    tokenizer: 'game_token_autocomplete',
    filter: ['lowercase', 'asciifolding']
  },
  custom_standanr: {
    type: 'custom',
    tokenizer: 'standard',
    filter: ['lowercase', 'asciifolding']
  },
  tags_analyzer: {
    type: 'custom',
    tokenizer: 'tags_tokenizer',
    filter: ['lowercase', 'asciifolding', 'trim']
  }
},
```

Figura 5: Código do analisador

### 3.3.2 Tokenizadores

Os tokenizadores são responsáveis por transformar os dados de maneira que possam ser eficientemente utilizados na indexação invertida. Neste projeto, empregamos três tipos de tokenização: a padrão do Elasticsearch, a *edge\_ngram* e uma personalizada. A seguir, explicaremos na prática como cada uma funciona:

**standard:** Esta abordagem de tokenização é baseada em gramática, utilizando o algoritmo de segmentação de texto Unicode. É particularmente útil para pesquisar termos localizados em diferentes posições do texto.

```
{
  "analyzer": "standard",
  "text": "The 2 QUICK Brown-Foxes jumped over the lazy dog's bone."
}
```

[ the, 2, quick, brown, foxes, jumped, over, the, lazy, dog's, bone ]

Figura 6: Tokenizador standard

**edge\_ngram:** Esta forma de tokenização divide o texto em n-gramas<sup>2</sup>. É particularmente eficaz em pesquisas que funcionam como auto-completar, oferecendo sugestões enquanto o usuário digita.

```
{
  "tokenizer": "standard",
  "filter": [
    { "type": "edge_ngram",
      "min_gram": 1,
      "max_gram": 3
    }
  ],
  "text": "the quick brown fox jumps"
}
```

[ t, th, the, q, qu, qui, b, br, bro, f, fo, fox, j, ju, jum ]

Figura 7: Tokenizador edge\_ngram

Por fim, implementamos um tokenizador personalizado para o nosso cenário, que é relativamente simples. Levando em conta os dados disponíveis, temos atributos como gêneros, categorias e idiomas. Estes dados não necessitam de previsão por parte do usuário, já que o sistema os apresenta de forma predefinida. Por isso, não há necessidade de um tratamento complexo, pois as consultas serão precisas. No entanto, é essencial aplicar uma tokenização específica para a maneira como essas informações estão estruturadas, geralmente como um texto contínuo nas características de um jogo. A imagem 8 ilustra essa particularidade:

```
languages: "English, French, Italian, German, Spanish - Spain, Japanese, Portugues..."
image: "https://cdn.akamai.steamstatic.com/steam/apps/655370/header.jpg?t=1617..."
windows: "True"
mac: "True"
linux: "False"
meta_score: 0
user_score: 0
positive: 53
negative: 5
recommendations: 0
developers: "Rusty Moyher"
publishers: "Wild Rooster"
categories: "Single-player, Steam Achievements, Full controller support, Steam Leaderb..."
genres: "Action, Indie"
```

Figura 8: Característica de exemplo

<sup>2</sup>Um n-grama é uma sequência de n letras adjacentes (incluindo sinais de pontuação e espaços em branco), sílabas, ou, em casos raros, palavras inteiras, encontradas em um conjunto de dados



**custom:** Este tokenizador personalizado utiliza um padrão específico encontrado nos atributos dos jogos, o símbolo de vírgula ”,”. Ele é projetado para separar os dados com base neste símbolo, facilitando a organização e a recuperação de informações.

```
{
  "tokenizer": "custom",
  "filter": {
    "type": "pattern",
    "pattern": ','
  },
  "text": "English,Japanese,Portuguese-Brasil,Spanish"
}
```

[English, Japanese, Portuguese-Brasil, Spanish]

Figura 9: Tokenizador customizado

### 3.3.3 Mapeando os atributos para indexação

Conforme mencionado anteriormente, optamos por dividir o sistema de forma a separar o sistema de recuperação de informações (RI) do sistema de gerenciamento de banco de dados (SGBD). Portanto, tornou-se necessário mapear quais características são essenciais para a recuperação das informações. Campos como screenshots e descrição foram considerados irrelevantes para este propósito.

```

mappings: {
  properties: {
    id: { type: 'text' },
    name: {
      type: 'text',
      analyzer: 'game_analyzer_complete',
      fields: {
        standard: {
          type: 'text',
          analyzer: 'custom_standanr'
        }
      }
    },
    image: {
      type: 'text',
      index: false
    },
    positive: {
      type: 'integer'
    },
    recommendations: {
      type: 'integer'
    },
    languages: {
      type: 'text',
      analyzer: 'tags_analyzer'
    },
    categories: { ...
    },
    genres: { ...
    },
    developers: { ...
    },
    publishers: { ...
    },
    release_date: {
      type: 'date',
      format: 'yyyy-MM-dd'
    },
    price: {
      type: "float"
    }
  }
}

```

Figura 10: Propriedades e mapeamento de indexação.

Vamos analisar a Figura 10. É possível observar que o campo 'name' utiliza dois analisadores que criamos: o 'standard' e o 'edge\_ngram'. Isso foi planejado para permitir que o usuário digite o nome do jogo e o sistema ofereça a função de auto-completar, além de possibilitar a busca por partes do nome, mesmo que em ordem incorreta, caso o usuário não se lembre do nome completo.

Os atributos como 'languages', 'categories', 'genres', 'developers' e 'publishers' são processados pelo analisador customizado com o padrão de tokenização que desenvolvemos, considerando que são atributos de correspondência exata. Ou seja, o usuário pesquisará exatamente pelo termo, que será fornecido pela interface do usuário.

Por último, há diversos atributos indexados para facilitar a filtragem, como data de lançamento, preço e recomendações.

### 3.3.4 Indexação e Inserção em Massa (Bulk Insert)

Após a geração dos tokens a partir dos atributos dos jogos, o Elasticsearch executa o processo de indexação. Ele emprega árvores balanceadas e busca binária para recuperar de forma eficiente os tokens e os documentos associados a eles.

Para inserir e indexar um documento no sistema, utilizamos o código a seguir:

```
const response = await elasticClient.index({
  index: 'jogos',
  document: {
    name: 'Nome do Jogo',
    genre: 'Ação',
    release_date: '2023-01-01',
  }
});

await elasticClient.indices.refresh({ index: 'jogos' }); // Para tornar o documento pesquisável imediatamente
```

Figura 11: Inserção de novo documento e re-indexação.

No entanto, para o nosso projeto, a indexação manual de cada documento não era a abordagem ideal. O que precisávamos era de um método que permitisse a inserção e indexação de documentos em massa. O Elasticsearch oferece um mecanismo específico para essa finalidade:

```
const cursor = await repository.getDb("steam").collection("games")
  .find()
  .project({
    name: 1,
    image: 1,
    positive: 1,
    recommendations: 1,
    languages: 1,
    categories: 1,
    genres: 1,
    developers: 1,
    publishers: 1,
    release_date: 1,
    price: 1
  })

const games = await cursor.toArray();

const bulkGames = games.map(game => ({
  id: game._id,
  name: game.name,
  image: game.image,
  positive: game.positive,
  recommendations: game.recommendations,
  languages: game.languages,
  categories: game.categories,
  genres: game.genres,
  developers: game.developers,
  publishers: game.publishers,
  release_date: game.release_date,
  price: game.price
})))

const operations = bulkGames.flatMap(doc => [{ index: { _index: 'jogos' } }, doc])

return await elasticClient.bulk({ refresh: true, operations })
```

Figura 12: Bulk insert e indexação.

## 3.4 Recuperando Dados

### 3.4.1 Rankeamento

Antes de explorarmos as diversas formas de filtrar jogos, é importante discutir o sistema de pontuação utilizado e as adaptações que implementamos para otimizar a busca de jogos.

Comumente, quando um usuário busca por um jogo, é essencial mostrar-lhe opções com boas avaliações e popularidade. Para atingir esse objetivo, desenvolvemos uma função de ranqueamento baseada em avaliações positivas.

O Elasticsearch oferece flexibilidade suficiente para personalizarmos o cálculo de pontuação. A Figura 13 a seguir ilustra essa customização. Atributos como `field`<sup>3</sup>, `factor`<sup>4</sup>, `modifier`<sup>5</sup>, `score_mode`<sup>6</sup> e `boost_mode`<sup>7</sup> são elementos que podemos modificar para ajustar nossa pontuação. Os valores utilizados foram o resultado de diversos testes, visando um sistema de busca estável que não apenas mostre ao usuário o que ele procura, mas também refine a busca para obter melhores resultados.

```
function_score: {
  query: {
    bool: bool
  },
  functions: [
    {
      ...name && {
        filter: {
          match: {
            name: {
              query: name as string,
            }
          }
        },
        field_value_factor: {
          field: "positive",
          factor: 0.001,
          modifier: "ln2p"
        }
      },
    },
  ],
  score_mode: "sum",
  boost_mode: "multiply"
},
```

Figura 13: Função de score personalizado baseado em notas positivas

### 3.4.2 Filtragem por Nome

A filtragem por nome é o método principal para recuperar jogos no sistema, recebendo, portanto, uma ênfase maior em termos de recursos disponíveis. Como já mencionamos anteriormente, essa filtragem utiliza dois analisadores, o que permite a busca por fragmentos de texto e também buscas em tempo real, conforme o usuário digita.

Adicionalmente, empregamos a técnica de fuzzing<sup>8</sup>, que possibilita buscas por aproximação, sendo ideal para casos de erros de digitação.

A seguir, apresentamos o código responsável por combinar esses dois analisadores. Além disso, incluímos uma imagem para ilustrar o conceito de fuzzing.

---

<sup>3</sup>Define qual campo do documento será utilizado.

<sup>4</sup>Multiplicador do atributo.

<sup>5</sup>Modifica o valor resultante do factor.

<sup>6</sup>Determina como a pontuação geral das consultas é calculada.

<sup>7</sup>Determina como a pontuação geral das funções de pontuação é calculada.

<sup>8</sup>A pesquisa difusa é uma técnica que emprega algoritmos de busca para encontrar strings que correspondam de forma aproximada a um padrão.

```

should: [
  {
    match: {
      "name.standard": {
        query: name as string,
        fuzziness: 1,
        boost: 5
      }
    }
  },
  {
    match: {
      name: {
        query: name as string,
        fuzziness: 1,
      }
    }
  }
]

```

Figura 14: Query para filtrar por nome

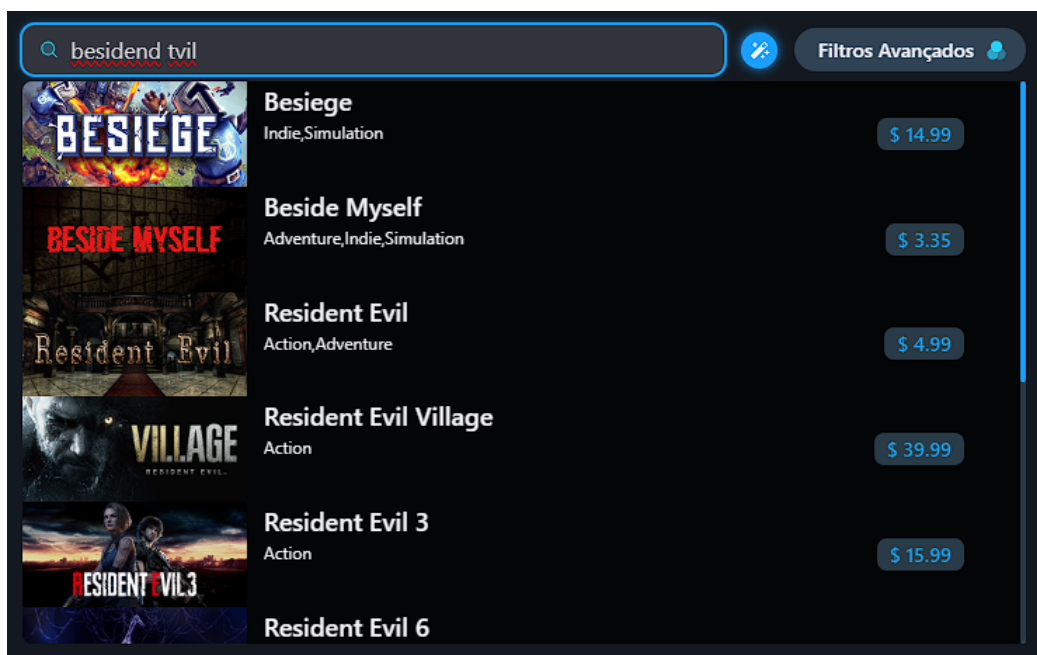


Figura 15: Exemplo de fuzzing

Como ilustrado na Figura 16, mesmo digitando de forma propositalmente errada a pesquisa, que deveria ser *resident evil* como *besidend tvil*, o sistema foi capaz de localizar os jogos correspondentes.

### 3.4.3 Filtragem Avançada

Além da filtragem por nome, nosso sistema também permite a combinação de filtros. Conforme explicado anteriormente, não é necessário aplicar analisadores complexos nem a técnica de fuzzing para esses casos, pois o cliente (usuário) já fornece os dados de forma parametrizada para a consulta.

```

...(filters.length) && {
  filter: filters,
  // filter = [
  //   match: {
  //     language: "English"
  //   },
  //   match: {
  //     language: "Spanish"
  //   },
  //   match:{
  //     categories: "Single-Player"
  //   }
  // ]
}

```

Figura 16: Filtragem por outros atributos

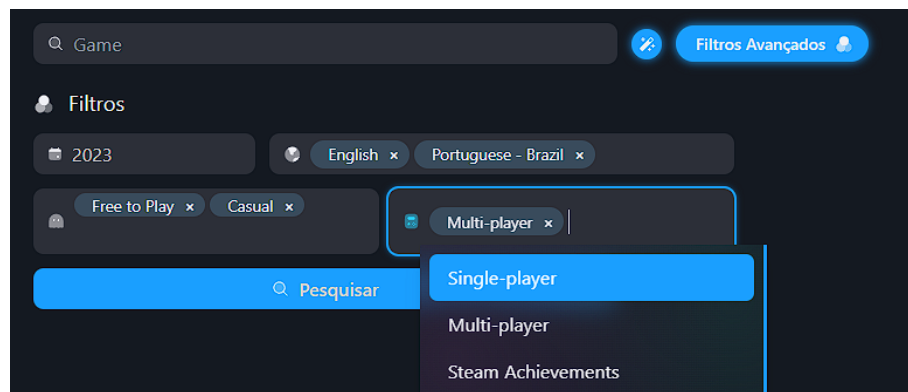


Figura 17: Exemplo de tela para filtragem complexa

Conforme mencionado anteriormente, a Figura 17 ilustra como os dados de tags, incluindo linguagem, categoria e gênero, são fixos para o usuário, eliminando a necessidade de analisadores complexos.

### 3.5 Inserindo Novo Documento

Dado que nosso sistema é dividido entre o sistema de gerenciamento de banco de dados (SGBD) e o sistema de recuperação de informações (RI), é necessário inserir os dados em ambos. Adicionalmente, para o novo dado, é preciso realizar o processo de indexação, que inclui as etapas de pré-processamento e tokenização. No Elasticsearch, uma vez que criamos um índice denominado "jogos", é suficiente utilizar a função `elasticCliente.index` para inserir um novo documento.

A Figura 18 a seguir apresenta um trecho de código responsável por essa lógica:

```

const resultado = await repository.getDb("steam")
  .collection("games")
  .insertOne(game)

if (!resultado.insertedId) res.status(500).json("ERROR")

try {
  const document = {
    id: String(resultado.insertedId),
    ...game
  }

  delete document._id

  const elasticResponse = await elasticClient.index({
    index: 'jogos',
    document: document
  })

  await repository.getDb("steam").collection("user")
    .insertOne({
      game: resultado.insertedId
    })

  return res.status(200).json(elasticResponse)
}
catch (e) {
  await repository.getDb("steam")
    .collection("games")
    .deleteOne({ _id: new ObjectId(resultado.insertedId) })

  return res.status(500).json("ERROR")
}

```

Figura 18: Inserção de novo documento em ambos sistema e indexação.

### 3.6 Desabilitando recursos do RI

Implementamos no sistema uma funcionalidade que permite comparar os resultados das buscas com e sem recursos como fuzzing, ranqueamento e tokens customizados. As figuras subsequentes ilustram essa diferença: na primeira, observamos uma busca realizada sem esses recursos, resultando na apresentação de jogos menos conhecidos e com avaliações positivas mais baixas. Na outra figura, o cenário é o oposto, destacando jogos mais populares e bem avaliados.

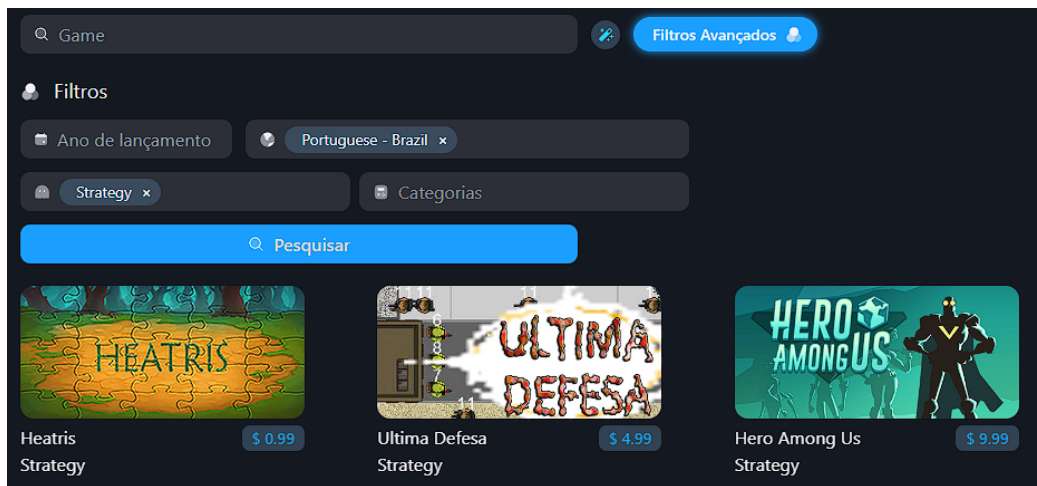


Figura 19: Exemplo sem 'mágica' do RI

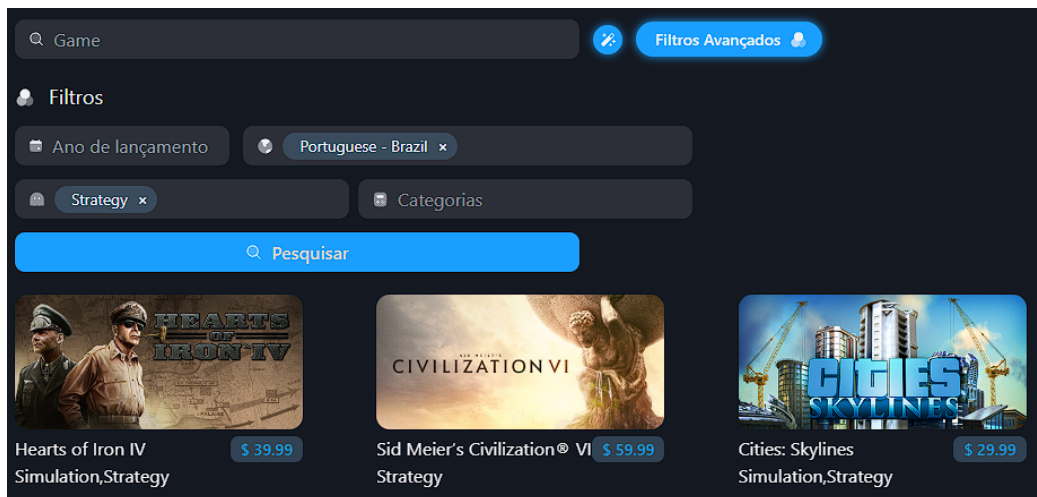


Figura 20: Exemplo com 'mágica' do RI

### 3.7 Performance

De modo geral, o sistema executa as buscas de forma muito rápida, considerando a complexidade das queries envolvidas. Quando não utilizamos o campo 'name', as respostas são mais ágeis, pois o custo computacional de utilizar fuzzy e múltiplos analisadores é significativo. Por outro lado, ao incluir este atributo, a média de tempo de resposta aumenta um pouco, mas sem ser algo expressivamente mais lento.

Status: 200 OK Size: 2.66 KB Time: 10 ms

Figura 21: Os tempos ficam abaixo de 20ms

Status: 200 OK Size: 4.42 KB Time: 37 ms

Figura 22: Os tempos geralmente abaixo de 40ms, com picos em 90ms



## 4 Exemplos

### 4.1 Pesquisa por nome

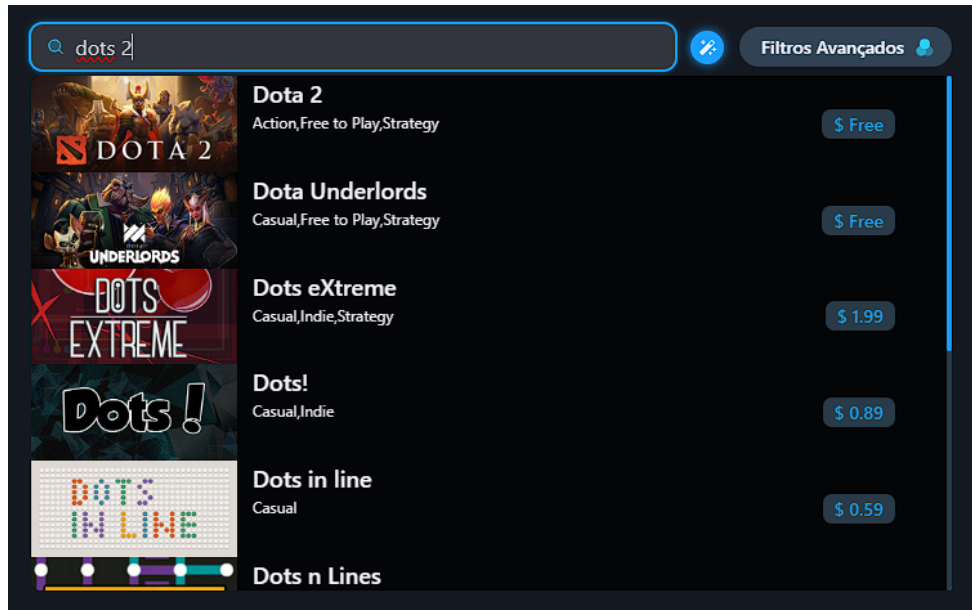


Figura 23: Exemplo de fuzzing e rankeamento. (Dota 2 é um jogo bem popular)

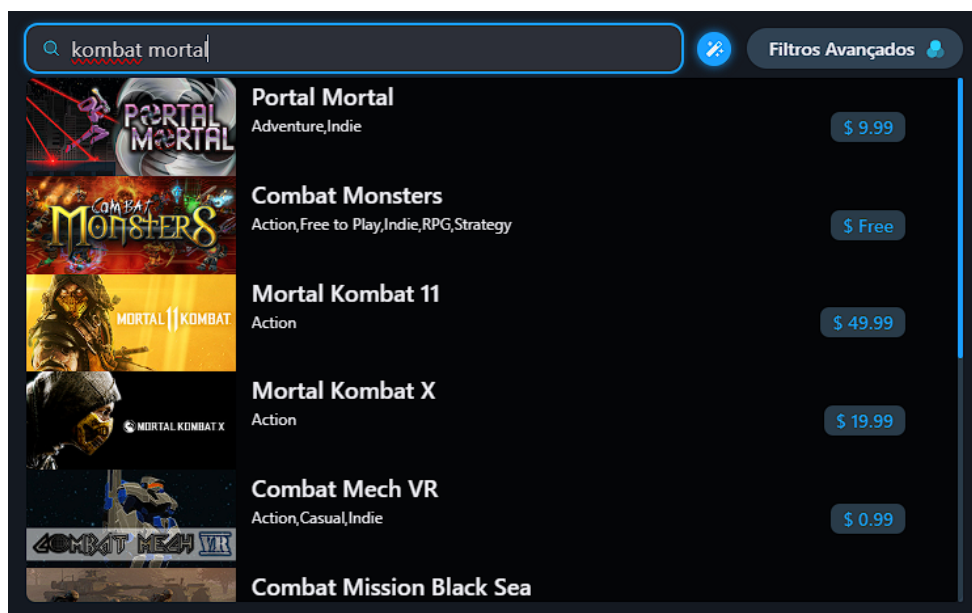


Figura 24: Exemplo da utilização do token standard.

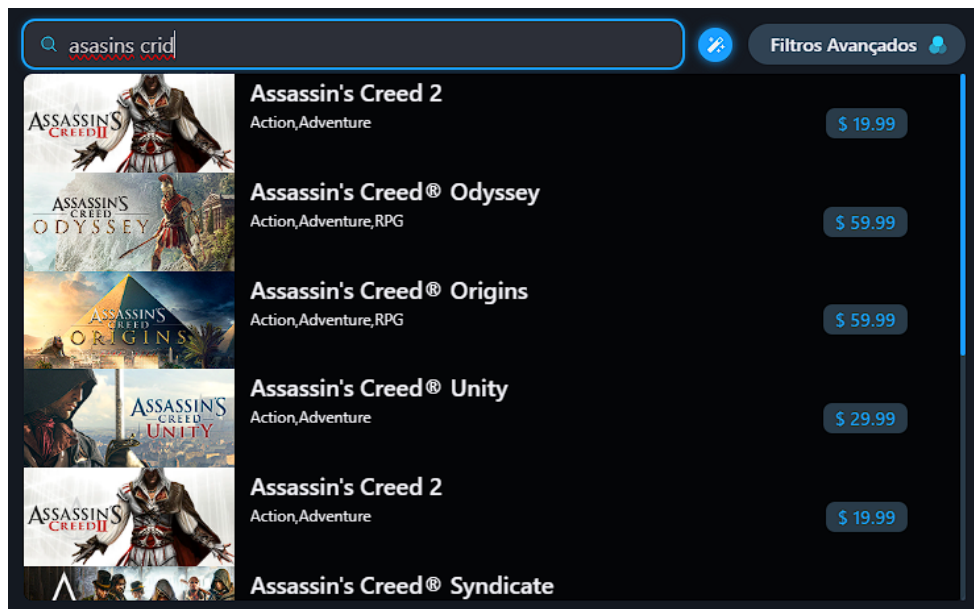


Figura 25: Exemplo de fuzzing.

## 4.2 Pesquisa avançada

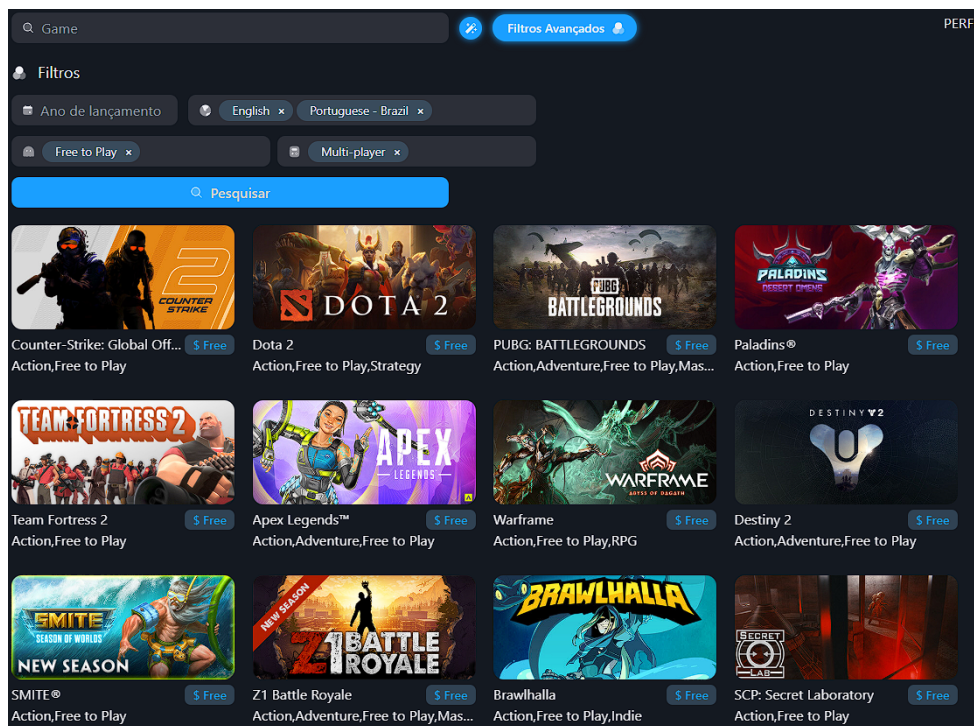


Figura 26: Exemplo de filtragem complexa.

### 4.3 Inserção de novo documento

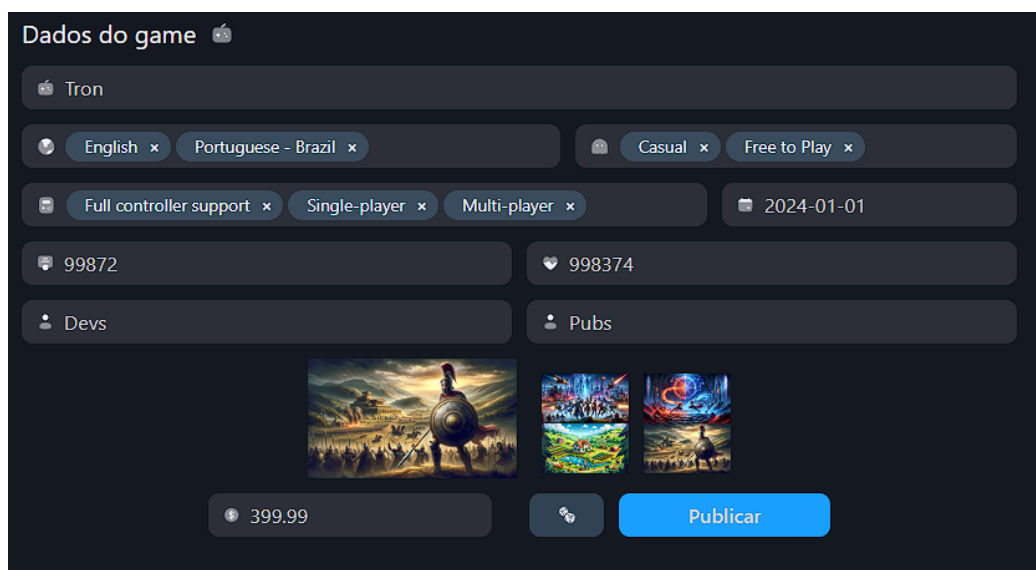


Figura 27: Exemplo de inserção de novo dado.

### 4.4 Visualização e exclusão de documentos inseridos

Os jogos inseridos podem ser pesquisados normalmente, assim como os outros jogos já presentes no sistema.

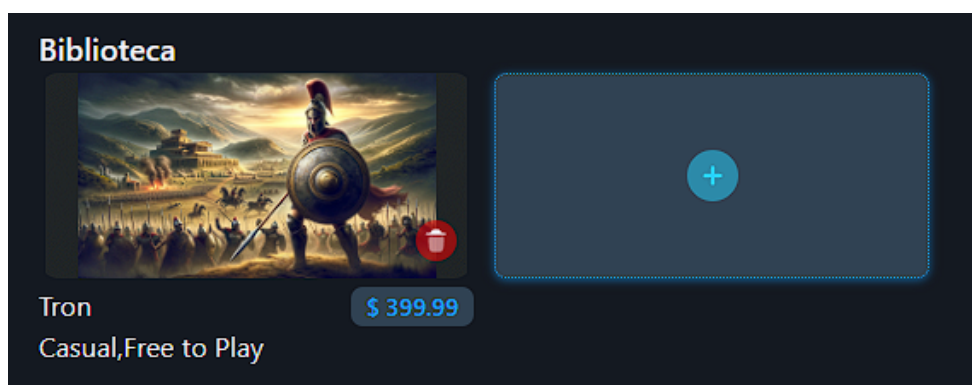


Figura 28: Exemplo de biblioteca de jogos adicionada.

## 5 Conclusão

O trabalho prático, concluído com êxito, abordou diversos conceitos discutidos em sala de aula e os aplicou de forma prática. O uso das ferramentas foi além da simples aplicação de seus recursos; buscamos entender como elas funcionam e como melhor aproveitá-las em nosso projeto.