Readme.pdf
Rafael Santos – 118336

PARTE 1

NOTA: Uma vez que realizei o trabalho sozinho (a colega que estava comigo desistiu da UC) utilizei, no SQLServer, a database com o meu número (sbd24_118336) e não as da "team".

Exercício 1 – Eliminação de Fontes Desnecessárias

O primeiro passo consistiu na remoção de registos associados a fontes consideradas não prioritárias para o contexto do trabalho.
As tabelas geneDiseaseNetwork e variantDiseaseNetwork continham registos provenientes das fontes *CLINVAR*, *UNIPROT* e *HPO*, que foram eliminados com comandos DELETE.
De seguida, foi utilizado o comando VACUUM para compactar o ficheiro da base de dados e refletir a libertação de espaço.
Após esta operação, o ficheiro apresentava um tamanho aproximado de 134 MB.

Exercício 2 – Análise do Tamanho das Tabelas

Com o intuito de identificar quais as tabelas mais pesadas, foi efetuada uma contagem do número de registos em cada tabela da base de dados.
As tabelas com mais de 100.000 registos e, por conseguinte, candidatas a intervenção, foram as seguintes:

- geneDiseaseNetwork (280.679 registos)
- variantAttributes (194.515 registos)
- variantDiseaseNetwork (157.786 registos)
- variantGene (172.427 registos)

Foi ainda analisada a estrutura destas tabelas através do comando PRAGMA table_info(...), com o objetivo de identificar atributos relevantes que permitissem uma limpeza criteriosa.

Exercício 3 – Limpeza da Base de Dados

Fase 1 – Limpeza inicial com critérios conservadores

Foram eliminados registos com valores reduzidos nos principais indicadores de relevância:

- Em geneDiseaseNetwork e variantDiseaseNetwork, foram removidos registos com score < 0.1
- Em variantAttributes, foram eliminadas variantes com DSI < 0.1 e DPI < 0.1
- Foram também apagadas as referências correspondentes em variantGene e variantDiseaseNetwork
- Finalmente, em variantDiseaseNetwork, eliminaram-se entradas com EI < 0.1

Esta fase permitiu uma redução significativa do tamanho da base, mas insuficiente para atingir os valores exigidos (ficando ainda acima de 80 MB).

Fase 2 – Limpeza aprofundada

De forma a reduzir ainda mais o tamanho da base de dados, procedeu-se a uma filtragem mais exigente:

- Eliminação de todos os registos em geneDiseaseNetwork com score < 0.4, uma medida mais abrangente mas ainda justificada
- Redução da tabela variantGene em aproximadamente 50%, eliminando as linhas onde geneNID ou variantNID eram números pares

Estas operações foram seguidas por nova compactação (VACUUM), reduzindo o tamanho para cerca de 61 MB.

Fase 3 – Eliminação de variantes com baixa especificidade ou impacto

Na fase final, foi aplicada uma filtragem adicional na tabela variantAttributes, com remoção de variantes cujo DSI ou DPI fosse inferior a 0.4.
As dependências em variantGene e variantDiseaseNetwork foram novamente eliminadas antes de apagar as variantes em si, de forma a garantir a integridade referencial.

Resultado Final

Após todas as fases de limpeza e compactação, o ficheiro final disgenet_2020.db ficou com 46,7 MB, satisfazendo o requisito de reduzir o tamanho do ficheiro para um valor entre 40 e 50 MB.
Todos os procedimentos foram efetuados com critério, respeitando as relações entre tabelas e garantindo a coerência da base de dados.

Exercício 4

4.1 – Criação do Esquema da Base de Dados no Microsoft SQL Server

Nesta etapa, foi necessário recriar no Microsoft SQL Server o esquema da base de dados anteriormente trabalhado em SQLite. Para isso, recorreu-se à funcionalidade do SQLiteStudio "Create similar table", que permite gerar as instruções DDL de cada tabela.

Foram geradas e revistas as instruções CREATE TABLE correspondentes às oito tabelas: diseaseClass, diseaseAttributes, disease2class, geneAttributes, geneDiseaseNetwork, variantAttributes, variantDiseaseNetwork e variantGene. Durante esta conversão, foi necessário ajustar os tipos de dados para garantir compatibilidade com o SQL Server, nomeadamente substituindo TEXT por VARCHAR(MAX) e ajustando tipos como TINYINT, SMALLINT, FLOAT e INT, de acordo com a natureza dos dados.

Também foram criadas todas as chaves primárias e chaves estrangeiras, assegurando a integridade referencial entre as tabelas, tal como definido no modelo relacional original.

4.2 – Exportação dos Dados em Ficheiros CSV

Com a base de dados limpa e estruturada, procedeu-se à exportação dos dados de cada tabela da base de dados SQLite para ficheiros CSV, utilizando o SQLiteStudio. Este processo foi repetido para as oito tabelas, assegurando que os ficheiros estavam corretamente formatados (valores separados por vírgulas, com delimitação de linhas adequada).

Foi confirmado que os ficheiros gerados incluíam os cabeçalhos das colunas, facilitando a correspondência correta dos dados no processo de importação para o SQL Server.

4.3 – Criação das Tabelas no SQL Server

Utilizando o SQL Server Management Studio (SSMS), foi executado o script com as instruções CREATE TABLE previamente adaptadas, criando todas as tabelas na base de dados SQL Server.

Foi verificada a estrutura das tabelas após a criação, garantindo que os nomes das colunas, os tipos de dados e os relacionamentos entre tabelas (chaves estrangeiras) estavam corretamente definidos e funcionais.

4.4 – Ajustes no Esquema Relacional no SQL Server

Neste exercício foram realizadas várias operações de adaptação ao esquema de base de dados DisGeNET, migrado de SQLite para SQL Server. O principal objetivo foi garantir a integridade referencial, ajustar os tipos de dados e corrigir inconsistências decorrentes da importação inicial. As ações concretas incluíram:

- Conversão de tipos de dados incompatíveis, como colunas originalmente varchar com valores numéricos (ex: pLI, DPI, DSI, EI, score), que foram convertidas para FLOAT após validação e limpeza dos dados.
- Alteração de tipos chave para SMALLINT ou INT, conforme apropriado, para garantir compatibilidade entre tabelas relacionadas.
- Recriação de chaves primárias que impediam alterações de tipo.
- Criação e aplicação de chaves estrangeiras (foreign keys) para reforçar a integridade referencial entre as várias tabelas (por exemplo, geneNID, variantNID, diseaseNID).
- Eliminação de registos inválidos em variantGene, que causavam conflitos ao criar foreign keys.
- Validação com queries de verificação (TRY_CAST) antes de alterações sensíveis.

No final, o esquema ficou completamente ajustado, consistente e com todas as ligações relacionais corretamente definidas segundo as boas práticas do SQL Server.

Exercício 5 - Consulta e Visualização dos Dados

Neste exercício foi criada uma view que reúne informação sobre doenças do tipo *phenotype*, apresentando os campos diseaseId, diseaseName e diseaseClassName. Para tal, realizou-se uma junção das tabelas diseaseAttributes, disease2class e diseaseClass, filtrando apenas as doenças classificadas como *phenotype*.

Em ambientes multiutilizador, a prática recomendada seria conceder permissões de acesso específicas à view a utilizadores distintos e criar sinónimos para facilitar o acesso a esses dados por parte desses utilizadores. No entanto, como este trabalho foi realizado individualmente, numa única conta de utilizador, não foi necessário configurar permissões adicionais nem criar sinónimos.

Optou-se por consultar diretamente a view criada, garantindo assim simplicidade e eficiência no processo, sem comprometer a segurança ou a integridade dos dados num contexto em que não existe partilha entre utilizadores.

Esta abordagem permite evidenciar a criação e utilização de views para organizar e filtrar informação relevante, bem como compreender os mecanismos de controlo de acessos que podem ser implementados em ambientes com múltiplos utilizadores.

Exercício 6 – Criação de uma View e Controlo de Acessos

Neste exercício, foi criada uma *view* com o objetivo de apresentar apenas os dados relevantes relacionados com doenças do tipo phenotype. A view inclui três atributos: diseaseId, diseaseName e diseaseClassName, os quais foram obtidos através de *joins*

entre as tabelas diseaseAttributes, disease2class e diseaseClass, aplicando uma condição de filtragem sobre o tipo de doença.

Embora o enunciado pedisse também para conceder o privilégio de SELECT a outro utilizador e pedir a criação de um *synonym* por esse utilizador, optei por não realizar essa parte, visto que estou a desenvolver o trabalho de forma individual. Ainda assim, a view foi criada e testada com sucesso através de um comando SELECT, verificando que devolve corretamente os registos esperados.
Este exercício reforça a utilidade das views como mecanismo de abstração e controlo de acesso a subconjuntos de dados mais sensíveis ou específicos.

Exercício 7 – Privilégios e controlo de acesso

Neste exercício, o objetivo foi preparar a base de dados para acessos futuros através de Python (por exemplo, usando Jupyter Notebook), garantindo que apenas os privilégios estritamente necessários fossem concedidos.

Como estou a trabalhar sozinho, com um único utilizador (sbd24_118336), que é também o proprietário (dbo) da base de dados, não foi necessário criar um novo utilizador nem conceder permissões adicionais. O utilizador atual já possui todos os privilégios para executar ações como SELECT sobre qualquer tabela ou view.

No entanto, incluí no script SQL os comandos GRANT comentados, que mostram como seriam atribuídas permissões de leitura (SELECT) a outro utilizador, caso fosse necessário partilhar o acesso de forma segura. Esta abordagem demonstra o conhecimento da sintaxe correta para controlo de acessos, mesmo que, neste contexto, os comandos não precisem de ser executados.

Exercício 8 — Melhoria de performance com índices

Neste exercício foi demonstrado como a criação de índices pode melhorar o desempenho de consultas SQL em bases de dados com grande volume de informação.
Foram apresentados dois exemplos distintos:

Exemplo 1: Índice sobre a coluna diseaseName
Criou-se um índice na coluna diseaseName da tabela diseaseAttributes para acelerar pesquisas por doenças com nome exato, como por exemplo 'Diabetes mellitus'. Antes do índice, a pesquisa percorria todos os registos (table scan), enquanto após o índice o SQL Server consegue localizar rapidamente os resultados (index seek), reduzindo o custo da operação.

Exemplo 2: Índice sobre a coluna geneName

Neste exemplo foi criado um índice na coluna geneName da tabela geneAttributes. Este índice beneficia consultas que fazem JOIN com a tabela geneDiseaseNetwork e filtram por nome do gene (BRCA1, por exemplo).
Tal como no exemplo anterior, o ganho de performance deve-se à redução de leituras desnecessárias em memória e disco.

Exercício 9 – Histograma de doenças por classe

Neste exercício, foi utilizado um ambiente Jupyter Notebook com Python para aceder remotamente à base de dados sbd24_118336 alojada no servidor SQL do professor. A conexão foi feita através da biblioteca pyodbc, com as credenciais atribuídas ao aluno.

O objetivo foi criar um histograma que mostra o número de doenças por classe (diseaseClassName). Para isso:

- Foi feita uma junção entre as tabelas:
    - diseaseAttributes (informação das doenças),
    - disease2class (ligação entre doenças e classes),
    - diseaseClass (nome das classes de doenças).
- A consulta agrupa os registos por diseaseClassName e conta quantas doenças existem em cada classe.
- O resultado foi carregado para um DataFrame com a biblioteca pandas.
- Por fim, foi utilizado matplotlib.pyplot para gerar o histograma, que permite visualizar claramente quais são as classes com maior número de doenças registadas.

O gráfico gerado confirma que algumas classes, como *Nervous System Diseases* ou *Congenital and Neonatal Disorders*, possuem um número significativamente maior de doenças associadas.

Exercício 10 – Evolução do número de registos por tipo de doença e ano

Neste exercício foi solicitado criar um gráfico que representasse a evolução temporal do número de registos na tabela variantDiseaseNetwork, diferenciados por tipo de doença (type) e ano de publicação (year).

Para isso:

- Utilizámos uma consulta SQL com JOIN entre as tabelas variantDiseaseNetwork e diseaseAttributes, de forma a obter o type de cada doença associada;
- Aplicámos um GROUP BY por year e type, contando o número total de registos por grupo;

- O resultado foi convertido para um DataFrame em Python e representado com matplotlib, usando um gráfico de linhas para mostrar a evolução de cada tipo de doença ao longo do tempo.

Este gráfico permite identificar tendências e picos na produção científica ou no registo de dados por tipo de doença, destacando, por exemplo, o aumento acentuado de fenótipos registados em determinados anos.

Exercício 11 — Exportação de Dados: Genes e Número de Entradas

Neste exercício foi solicitado criar um ficheiro Excel (ou CSV) contendo uma tabela com os campos geneId, geneName e o número de entradas correspondentes na tabela geneDiseaseNetwork.

Para isso, foi usada a linguagem Python no ambiente Jupyter Notebook com uma query SQL que:

- Agrupa os registos da tabela geneDiseaseNetwork por geneId e geneName
- Conta o número de entradas por gene
- Ordena os resultados de forma decrescente pelo número de entradas

O resultado foi exportado para um ficheiro CSV usando pandas.to_csv() com index=False, garantindo que o ficheiro não incluía colunas adicionais desnecessárias.

O ficheiro final chama-se gene_entries.csv e cumpre o objetivo da tarefa.

Exercício 12 – Pesquisa de doenças associadas a um gene

Neste exercício foi desenvolvido um pequeno script em Python (utilizado no Jupyter Notebook) com o objetivo de permitir a pesquisa de doenças associadas a um gene, a partir do geneId ou do geneName.

O utilizador é questionado sobre o tipo de pesquisa que pretende realizar (id ou name) e, com base na sua escolha, é executada uma query SQL que retorna os seguintes dados:

- diseaseNID
- diseaseId
- diseaseName
- type (tipo da doença)

Foram criadas duas instruções SQL parametrizadas, de forma a proteger a aplicação contra injeções SQL, e utilizadas com o método pandas.read_sql(). O resultado é apresentado sob a forma de tabela (DataFrame), facilitando a análise no ambiente do Jupyter Notebook.

Além disso, o código foi preparado para:

- Detetar e informar o utilizador se não existirem resultados.
- Lidar com entradas inválidas (como digitar algo diferente de "id" ou "name").
- Fechar corretamente a ligação à base de dados.

Este exercício simula uma funcionalidade muito comum em aplicações de base de dados acessadas remotamente, reforçando o uso de interfaces simples e eficazes em Python para explorar dados clínicos e genéticos.

Exercício 13 – Inserção de dados na tabela diseaseAttributes

Neste exercício foi desenvolvido um script em Python com o objetivo de inserir novos registos na tabela diseaseAttributes da base de dados SQL Server.

O script realiza as seguintes operações:

1. Estabelece ligação com a base de dados utilizando as credenciais fornecidas pelo professor;
2. Obtém automaticamente o próximo valor disponível para o campo diseaseNID, garantindo que não há duplicações;
3. Solicita ao utilizador os restantes campos necessários:
    - diseaseId (exemplo: C0000729)
    - diseaseName (exemplo: Abdominal Cramps)
    - type (exemplo: phenotype);
4. Executa uma query parametrizada INSERT INTO, assegurando proteção contra injeções SQL;
5. Confirma a operação com conn.commit();
6. Em caso de erro, apresenta uma mensagem informativa sem quebrar a execução.

A entrada de dados é feita de forma interativa, com exemplos práticos apresentados ao utilizador para maior clareza. Esta abordagem garante robustez, segurança e facilidade de utilização ao realizar inserções manuais na base de dados.

Exercício 14 – Validação da coluna type com chave estrangeira

Neste exercício, foi necessário garantir que os valores inseridos na coluna type da tabela diseaseAttributes fossem válidos. Para isso, foi adotada uma abordagem de normalização e controlo de integridade referencial:

Criação da tabela diseaseTypes

Foi criada uma nova tabela chamada diseaseTypes, que armazena todos os valores distintos e válidos encontrados na coluna type da tabela original diseaseAttributes. Esta tabela serve como base de validação.

Criação da tabela diseaseAttributes_v2

Em seguida, criou-se uma nova tabela diseaseAttributes_v2, com estrutura semelhante à original, mas com uma chave estrangeira (foreign key) que referencia a tabela diseaseTypes. Esta FK garante que apenas valores permitidos podem ser inseridos na coluna type.

Foram então migrados os dados válidos da tabela original para esta nova tabela, respeitando a integridade dos dados.

Adaptação do script Python

O script desenvolvido no exercício 13 foi modificado para funcionar com a nova tabela. Inclui agora:

- A listagem automática dos tipos válidos disponíveis;
- Uma verificação do valor introduzido pelo utilizador;
- A rejeição da inserção com mensagem de aviso se o tipo for inválido;
- A atribuição automática do próximo diseaseNID.

Este processo reforça a robustez e qualidade dos dados da base de dados.

Exercício 15 – Acesso remoto a dados meteorológicos via API (WeatherAPI)

Neste exercício foi desenvolvido um script em Python com o objetivo de aceder a dados meteorológicos em tempo real, utilizando a API pública da WeatherAPI. A cidade selecionada para consulta foi Aveiro, Portugal.

Lógica do Código

O script começa por importar o módulo requests, essencial para realizar pedidos HTTP em Python. Em seguida:

1. Configuração da API:
    - Define-se a *API key* pessoal (8232784e010c41a7aa003833250806);
    - Especifica-se o URL base do serviço (http://api.weatherapi.com/v1/current.json);

    - Indica-se a cidade a consultar: "Aveiro, Portugal".
2. Construção do pedido:
    - Cria-se o URL final juntando o endpoint, a chave e o nome da cidade.
3. Pedido HTTP:
    - É feito um pedido GET ao serviço;
    - Se a resposta tiver sucesso (status code 200), os dados meteorológicos são extraídos da resposta em formato JSON.
4. Dados extraídos:
    - Latitude e longitude da cidade;
    - Descrição do estado do tempo (ex: "Partly cloudy");
    - Temperatura atual em graus Celsius;
    - Percentagem de humidade;
    - Velocidade do vento em km/h.
5. Apresentação dos resultados:
    - As informações são impressas no ecrã com formatação clara e organizada.

```
[Running] python -u "c:\Users\rafae\OneDrive\Ambiente de Trabalho\Ex_15\weatherGET_partial.py"
Weather in Aveiro, Portugal (lat:40.6333, lon:-8.65):
Description: Clear
Temperature: 12.0 Celsius
Humidity: 92%
Wind Speed: 4.0 km/h
```

Exercício 16 - MyGene.info

Pergunta 16.1 – Dados do gene com ID 593

Objetivo: Obter informação detalhada sobre o gene com o identificador 593.

Solução:

- Foi feita uma requisição ao endpoint: https://mygene.info/v3/gene/593.
- A resposta JSON foi processada para extrair e apresentar:
    - symbol: símbolo do gene.
    - name: nome descritivo.
    - summary: resumo da função do gene (caso disponível).
- Foi incluído tratamento para ausência do campo summary.

```
 Dados do gene 593:
Symbol: BCKDHA
Name: branched chain keto acid dehydrogenase E1 subunit alpha
Summary: The branched-chain alpha-keto acid (BCAA) dehydrogenase (BCKD) complex is an innter mitochondrial enzyme complex th
```

Pergunta 16.2 – Pesquisa pelo símbolo BRCA1

Objetivo: Procurar informações sobre o gene associado ao símbolo BRCA1.

Solução:

- Foi utilizada a URL: https://mygene.info/v3/query?q=symbol:BRCA1.
- A resposta contém uma lista de resultados no campo hits.
- O script extraiu os dados do primeiro resultado, mostrando:
    - symbol, name, _id (Entrez Gene ID), taxid (organismo) e summary.

```
 Informação sobre o gene BRCA1:
Symbol: BRCA1
Name: BRCA1 DNA repair associated
Entrez Gene ID: 672
Organism: 9606
Summary: Sem resumo disponível.
```

Pergunta 16.3 – Genes com as palavras "tumor" e "suppressor"

Objetivo: Escrever uma URL que recupere genes com as palavras "tumor" e "suppressor" em qualquer campo da base de dados.

URL utilizada: https://mygene.info/v3/query?q=tumor+AND+suppressor

Solução:

- A API retornou uma lista de genes com esses termos em qualquer atributo.
- O script extraiu e apresentou os 5 primeiros resultados, com:
    - symbol, name e summary (ou mensagem alternativa se ausente).

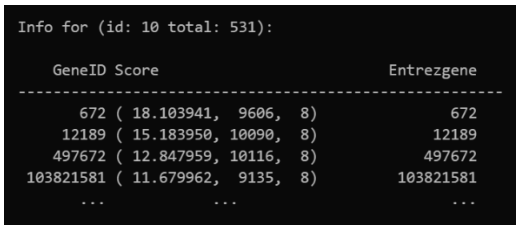
```
 Genes que contêm 'tumor' e 'suppressor':

Symbol: LNCPTCTS
Name: lncRNA papillary thyroid carcinoma tumor suppressor
Summary: Sem resumo disponível.
--------------------------------------------------
Symbol: LOC282551
Name: Insulinoma tumor suppressor gene locus
Summary: Sem resumo disponível.
--------------------------------------------------
Symbol: TSG11
Name: Tumor suppressor gene on chromosome 11
Summary: Sem resumo disponível.
--------------------------------------------------
Symbol: LNCPTCTS
Name: lncRNA papillary thyroid carcinoma tumor suppressor
Summary: Sem resumo disponível.
--------------------------------------------------
Symbol: MLRL
Name: Myeloid leukemia-related gene (myeloid tumor suppressor)
Summary: Sem resumo disponível.
--------------------------------------------------
```

Exercício 17 – Apresentação Formatada de Resultados da API MyGene.info

Nesta questão foi desenvolvida uma função em Python chamada mostrar_resultados_formatados(data), que recebe como entrada a resposta JSON da API MyGene.info (como na questão 16.2) e apresenta os dados dos primeiros genes retornados num formato tabular formatado.

A função imprime:

- Um cabeçalho com o número de resultados (id) e o total da pesquisa;
- Uma tabela com três colunas: GeneID, Score e Entrezgene;
- O campo Score é composto por três valores: o score da API (com 6 casas decimais), o taxID do organismo e o número fixo 8, tudo apresentado com espaçamento fixo para que os valores fiquem visualmente alinhados entre si;
- Os campos GeneID e Entrezgene estão alinhados à direita;
- No final, uma linha com reticências (...) é usada para indicar que existem mais resultados além dos apresentados.



Exercício 18 – Consulta Detalhada de Genes

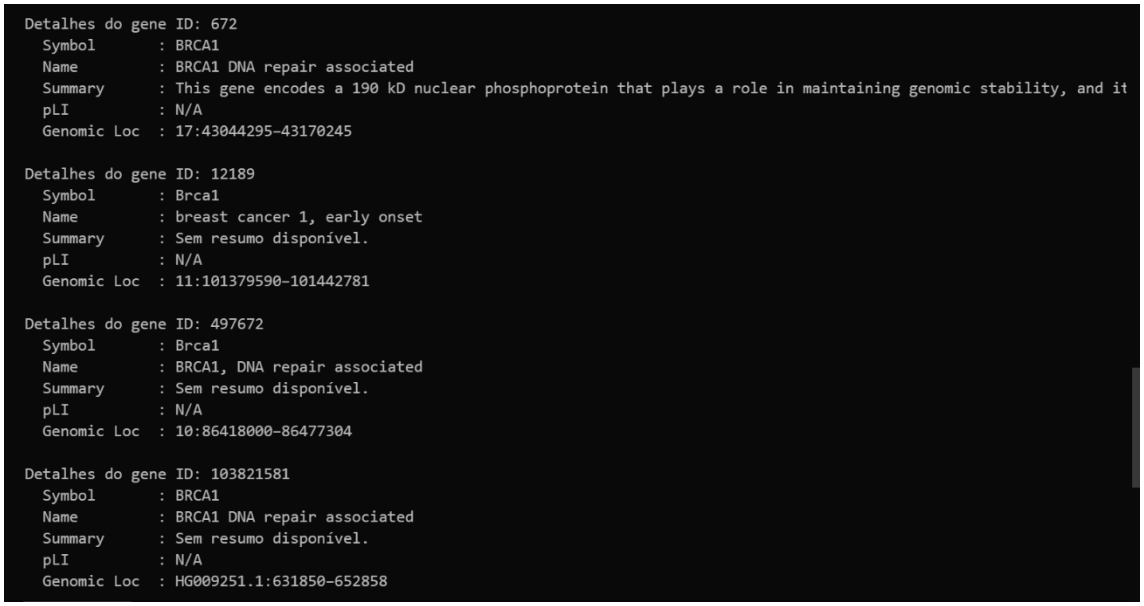
Foi criada a função detalhes_gene(gene_ids) para receber uma lista de geneIDs e, para cada um deles, fazer uma requisição à API https://mygene.info/v3/gene/<gene_id>.

O código processa os dados recebidos no formato JSON e extrai os seguintes campos:

- symbol
- name
- summary
- pLI
- genomic_loc

Foi implementado tratamento para diferentes formatos do campo genomic_pos, que pode vir como dicionário ou lista. Quando presente, a localização genómica é apresentada no formato <cromossoma>:<start>–<end>. Quando algum campo está ausente, é apresentada uma indicação apropriada ("N/A" ou "Sem resumo disponível.").

A informação de cada gene é apresentada no terminal de forma legível, organizada e indentada, com separação clara entre genes. A função foi testada com os mesmos IDs usados na questão 17 e executou corretamente para todos os casos.



Exercício 19 – Inserção de Dados em geneAttributes a partir da API MyGene.info

Nesta questão, foi implementado um processo completo de extração e inserção de dados gene annotation usando a API pública MyGene.info e a base de dados SQL Server fornecida.

O trabalho foi dividido em duas partes:

Parte 1 – Obtenção de Genes da API

Foi feita uma requisição à API https://mygene.info/v3/query com o termo de pesquisa "DNA" e limite de 15 resultados. O código extraiu os campos _id, symbol e name para cada gene retornado e apresentou-os de forma organizada no terminal, permitindo visualizar os genes que seriam inseridos a seguir.

Os IDs dos genes foram armazenados automaticamente numa lista (gene_ids) para utilização na segunda fase.

Parte 2 – Inserção na Tabela geneAttributes

Para cada gene da lista:

- Foi feita uma nova requisição à API para obter os dados detalhados (symbol, summary, pLI).

- O campo geneDescription foi construído com base no summary (ou name se o primeiro estiver ausente) e limitado a 255 caracteres para respeitar as restrições da base de dados.
- Foi gerado automaticamente um geneNID sequencial através de uma consulta MAX(geneNID) + 1.
- Os dados foram inseridos na tabela dbo.geneAttributes usando uma query parametrizada para segurança e compatibilidade.

Mensagens no terminal confirmaram a inserção bem-sucedida de todos os genes. Erros de truncamento foram resolvidos ao aplicar corte da string de descrição diretamente no código Python.

Resultado Final

15 genes foram inseridos com sucesso na tabela geneAttributes, com todos os campos relevantes preenchidos: geneId, geneName, geneDescription e pLI.

O código foi devidamente testado, modularizado e comentado, garantindo robustez e clareza.