Trabalho para conclusão de disciplina

Codificação para Armazenamento de Dados

Introdução

Este documento tem por objetivo indexar os estudos realizados pelo grupo composto por Ana Lívia, Ana Priss, Aline Mara, João Squinelato, Marcelo Pena e Thais Carvalho, como conclusão para a disciplina de Ecossistemas de Big Data, código eEDB-006/2024-2, ministrada pelo professor Luis Flávio da Silva.

Mais especificamente, o grupo propôs-se a se aprofundar sobre o tema “Codificação para armazenamento de dados”, baseando-se no capítulo 4, intitulado *Encoding and Evolution*, do livro *Designing data-intensive applications*, escrito por Kleppmann (2017) e publicado pela O’Reilly.

Prosseguindo, o documento se inicia com uma fundamentação teórica a respeito do cenário que aplicações computacionais se inserem; ressaltando a coexistência não só de versões antigas e novas de uma mesma aplicação, mas também dos dados gerados e, ou, consumidos por esta aplicação em seu cotidiano.

Posteriormente, o conceito de codificação é definido e são apresentados dois tipos de codificação: textual, utilizando o JSON, XML e CSV como exemplos e objetos de discussão e; binária, com o Apache Avro e Parquet.

Ademais, é apresentada a metodologia utilizada pelo grupo para a realização de testes práticos quanto a utilização de diferentes codificações para o armazenamento de dados. Logo em seguida, os resultados destes testes são apresentados e discutidos, os quais podem ser replicados acessando o repositório github mdspena/eEDB-006-2024-2.

Subsequentemente, uma conclusão a respeito dos diferentes tipos de codificação e a implicação de suas preferências no que tange ao armazenamento de dados é desenvolvida. Enfim, ao final do documento encontra-se uma lista com as referências bibliográficas utilizadas neste trabalho.

Fundamentação Teórica

Mudanças em aplicações computacionais são constantes. Em outras palavras, novas funcionalidades são adicionadas, requisitos são mais bem definidos, ou o contexto de negócios pode ser alterado. Na maioria das vezes, essas mudanças se refletem nos dados que uma aplicação computacional armazena, quer seja num contexto relacional ou não.

Quanto aos bancos de dados relacionais, tem-se a migração de esquemas, a qual força um esquema para cada ponto no tempo. Já no contexto de bancos de dados NoSQL, com o esquema sendo determinado no momento da leitura dos dados, há a coexistência de antigos e novos formatos, escritos em diferentes tempos, em um mesmo banco de dados.

Além disso, uma mudança no esquema dos dados, corresponde, normalmente, a uma mudança no código da aplicação computacional. Contudo, nem sempre estas mudanças podem ser feitas em tempo hábil, ou então refletidas em todas as instâncias da aplicação computacional. Isto quer dizer que versões antigas e novas do código da aplicação computacional, assim como dados em formatos antigos e novos, coexistem.

Logo, para garantir que a aplicação computacional se mantenha funcional, é necessário garantir duas coisas: (i) compatibilidade retroativa, isto é, o novo código deve interpretar dados em formatos antigos e; (ii) compatibilidade futura, o código antigo deve interpretar dados em formatos novos.

Codificação dos Dados

Rotineiramente, aplicações computacionais manipulam dados de, pelo menos, duas formas:

- Em memória, por meio de estruturas otimizadas para acesso eficiente e manipulação da CPU - normalmente por meio de ponteiros -, tais como, objetos, *structs*, listas, *arrays*, *hash tables*, árvores etc;
- Em disco ou via Internet, em que é necessário formatar os dados em uma sequência de *bytes*, uma vez que utilizar ponteiros em outra aplicação computacional perderia o sentido.

Isto posto, à essa tradução entre a representação em memória para a sequência de *bytes*, dá-se o nome de codificação, ou serialização, em que o reverso é chamado de decodificação, ou deserialização.

Assim, uma vez que a compatibilidade retroativa e futura em aplicações computacionais é um problema recorrente, existem diversas soluções disponíveis para se codificar os dados. Com isso, nas seções seguintes algumas destas soluções serão apresentadas.

JSON, XML e CSV

Começando com as codificações mais conhecidas, e utilizadas, em aplicações computacionais, JSON, XML e CSV são codificações textuais e, até certo grau, humanamente inteligíveis. Além disso, têm suporte para um grande conjunto de plataformas

e linguagens de programação, destacando-se o JSON com seu suporte nativo em navegadores de Internet, em virtude de originar-se da linguagem de programação JavaScript.

No entanto, tais facilidades trazem consigo alguns problemas. Um deles diz respeito à ambiguidade em interpretar números, isto é, em XML e CSV não há distinção entre um número e uma *string* que consista de números. Este tipo de problema não ocorre em JSON, porém não há distinção entre números inteiros e ponto flutuante.

Para mitigar tais ambiguidades, tanto XML quanto JSON permitem a adoção de esquemas de dados, os quais exigem uma curva de aprendizado e adaptação nas aplicações computacionais. Em contrapartida, o mesmo não pode ser dito em relação ao CSV, pois este não possui suporte ao uso de esquemas de dados.

Além disso, JSON e XML não têm suporte nativo para representar *strings* binárias, as quais são utilizadas para representar mídias como fotos ou áudios. Para lidar com tal limitação, normalmente utiliza-se a decodificação em base 64, que consiste em representar *strings* binárias por meio de 64 caracteres, o que aumenta o tamanho da *string* binária em 33%.

Apesar das codificações textuais necessitarem de maior espaço de armazenamento, sua inteligibilidade, popularidade e consenso - para com a troca de informações entre empresas e serviços - garante que estas codificações sejam padrão para muitas aplicações computacionais.

Assim, para elucidar as codificações textuais, abaixo tem-se a representação de um usuário e seus dados (Listagem 1), os quais foram codificados em JSON, consumindo 81 *bytes* para serem armazenados, desconsiderando espaços em branco, onde cada caracter consome 1 *byte*:

```
{
    "userName": "Martin",
    "favoriteNumber": 1337,
    "interests": ["daydreaming", "hacking"]
}
```

Listagem 1: Exemplo de JSON

Serializações Binárias

Existem cenários onde a pressão pela utilização de codificações padrão são menores. Por exemplo, com dados internos de uma instituição ou empresa, ou então caso se deseje economizar com o armazenamento deste dados, ou até decodificá-los com maior agilidade. Nesse contexto, surgem as codificações binárias, as quais atendem às demandas elencadas anteriormente em detrimento da inteligibilidade humana.

Apache Avro

Dentre as várias codificações binárias, incluindo às disponíveis para JSON e XML, destaca-se o Avro. Seu surgimento se deu após as codificações binárias existentes não se adequarem ao ecossistema do Hadoop, culminando no subprojeto Apache Avro.

Obrigatoriamente, para especificar a estrutura dos dados a serem codificados e decodificados, o Avro se utiliza de um esquema, o qual pode ser escrito por meio de uma linguagem de definição de interface (IDL), vide Listagem 2.

```
record Person {
    string               userName;
    union { null, long } favoriteNumber = null;
    array<string>        interests;
}
```

Listagem 2: IDL Avro representando os dados de uma pessoa

Então, considerando os mesmos dados contidos no JSON da seção anterior, e codificando-o para Avro, com definição do IDL acima, tem-se como resultado uma sequência de apenas 32 *bytes*.

Mais especificamente, na sequência de *bytes* resultante, não há qualquer indicativo sobre o nome ou o tipo dos campos, mas sim uma concatenação de valores hexadecimais precedidos por seu tamanho, vide figura abaixo.

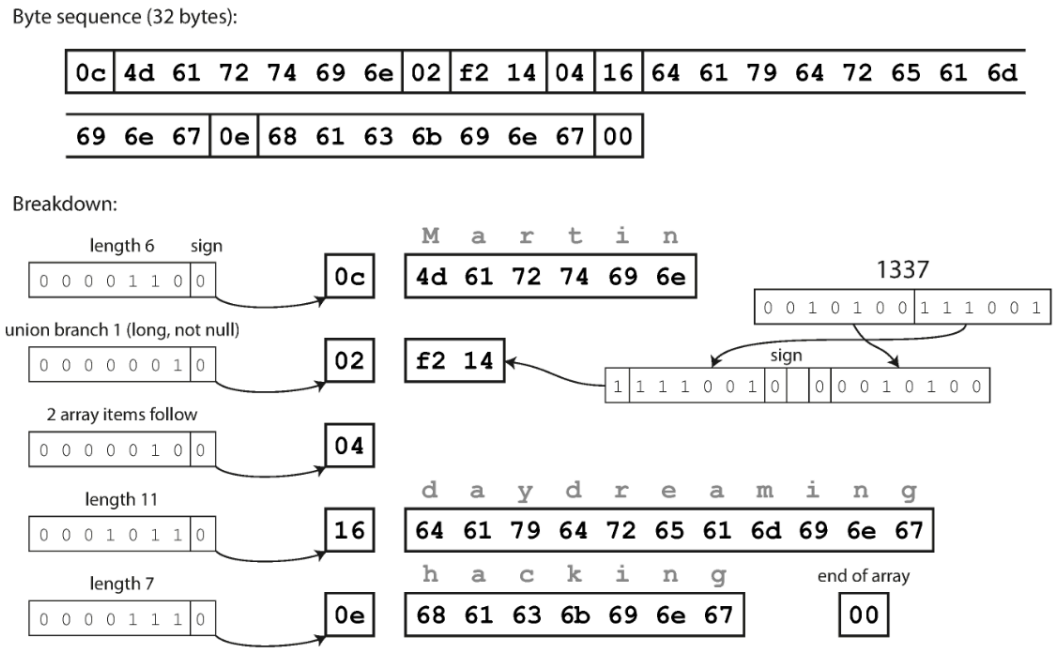

Figura 1: detalhamento sobre funcionamento da codificação binária Apache Avro

Logo, é evidente a necessidade de possuir um esquema tanto para escrever dados em Avro, quanto para lê-los. Contudo, o esquema de escrita e o esquema de leitura não têm de ser iguais, e sim compatíveis. Por exemplo, caso o esquema de leitura encontre campos que existam apenas no esquema de escrita, estes campos podem ser ignorados. Outrossim, se o esquema de leitura espera por campos que não existem no esquema de escrita, estes campos são preenchidos com valores padrão.

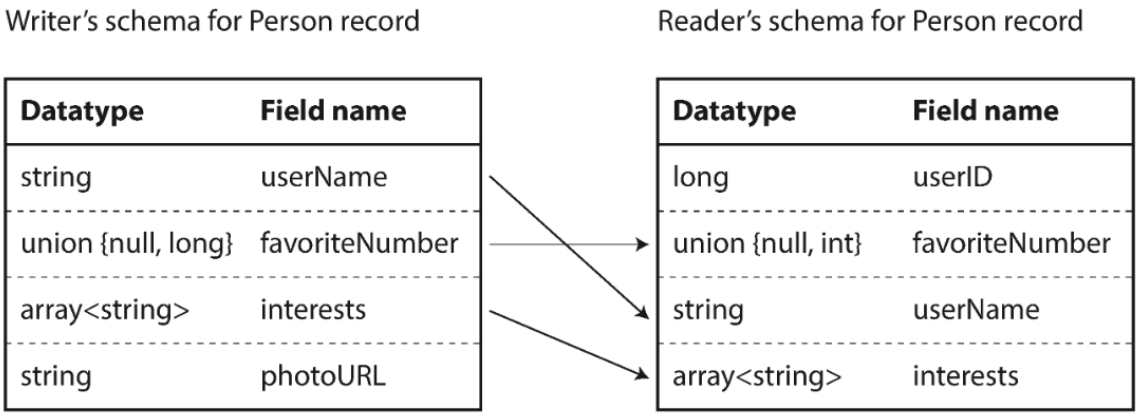

Figura 2: Comparação entre esquema de escrita e de leitura para Apache Avro

Outra vantagem da adoção de esquemas de dados está na capacidade natural de gerar uma documentação dos dados, assim como a evolução destes esquemas permite a mesma flexibilidade observada em bancos de dados NoSQL.

Apache Parquet

O Apache Parquet é uma codificação orientada à coluna, desenvolvido para armazenar dados de maneira eficiente e retorná-los de maneira rápida. Além disso, o Parquet utiliza dicionários de dados e algoritmos de compressão para reduzir ainda mais o espaço de armazenamento, assim como faz uso de metadados e esquemas de dados para garantir maior eficiência de leitura. Ademais, o Parquet é uma solução disponível para diferentes linguagens de programação e ferramentas de análise (DATABRICS, 2024).

Hierarquicamente, em um arquivo Parquet há um ou mais grupos de linhas (*row groups*), os quais contêm exatamente uma porção de coluna (*column chunk*) para cada coluna; as quais são compostas por páginas (pages), que por sua vez são unidades indivisíveis dos dados (DATABRICS, 2024). Isto posto, os detalhes dessa hierarquia podem ser observados com a Figura 3.

Mais especificamente, o Parquet armazena metadados das páginas, como informações dos valores únicos, valor máximo e mínimo para colunas numéricas, dentre outras informações. Além disso, existem metadados a nível de porções de colunas, assim como de grupos de linhas, os quais são armazenados no rodapé (*footer*) de um arquivo Parquet (BRAAMS, 2019).

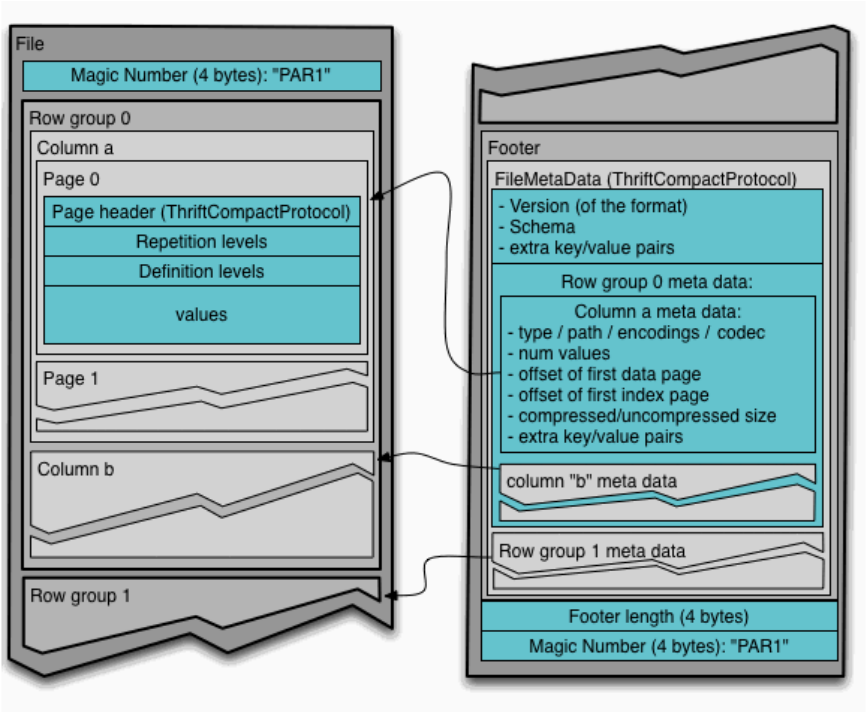

Figura 3: Hierarquia da Codificação Parquet

A vantagem de se utilizar estes metadados está na hora de realizar consultas em arquivos Parquet, onde especifica-se uma condição de filtro para os dados. Por exemplo, caso deseje-se saber o nome dos trabalhadores de uma empresa com salário igual ou superior a cinco mil reais, é ideal que seja feito o menor número de leituras, retornando apenas a coluna de interesse, isto é, o nome dos funcionários (BRAAMS, 2019).

Para garantir, então, uma melhor performance de busca para o problema levantado anteriormente, o Parquet se utiliza dos metadados, e de informações sobre os dados armazenados, tais como o valor máximo e mínimo de cada coluna, para ler apenas os grupos de linhas que contêm dados de interesse (BRAAMS, 2019).

Isto é, caso exista algum grupo de linhas cujo valor máximo para a coluna salário seja de quatro mil reais, sabe-se com segurança que este grupo de linhas pode ser ignorado seguramente no que se diz respeito ao problema levantado. Ademais, em virtude do Parquet ser um formato colunar, após filtrar-se os dados pela coluna salário, pode-se ler apenas a coluna nome, sem a necessidade de ler as demais colunas (BRAAMS, 2019).

Ainda, a adoção de metadados possui outra vantagem: a redução no armazenamento dos dados. Diferentemente do Avro, o qual armazena os valores e seus tamanhos, o Parquet se utiliza de um dicionário de dados, em que apenas índices que representam valores são armazenados, ao invés do próprio valor. Além disso, quando há repetições sequenciais de índices, armazena-se somente o número de repetições em sequência deste índice, o que reduz significativamente o armazenamento (BRAAMS, 2019).

Por fim, para reduzir ainda mais o espaço de armazenamento, o Parquet comprime esses índices de dados com algoritmos como o SNAPPY, GZIP, dentre outros. Isso não só reduz o espaço de armazenamento, mas também o tempo de leitura dos dados (BRAAMS, 2019).

Metodologia

Com o intuito de realizar testes práticos e delimitar concretamente as diferenças entre as codificações de dados apresentados neste estudo, foram utilizados dados públicos do Censo da Educação Superior fornecidos pelo Instituto Nacional de Estudos e Pesquisas Educacionais Anísio Teixeira (Inep).

Ademais, um ambiente de *Data Lake* foi construído utilizando a AWS como provedora de computação em nuvem. Mais especificamente, o grupo baseou-se na infraestrutura discutida no artigo *Build a Data Lake Foundation with AWS Glue and Amazon S3*, assinado por Heinrich (2017), a qual é ilustrada pela figura abaixo.

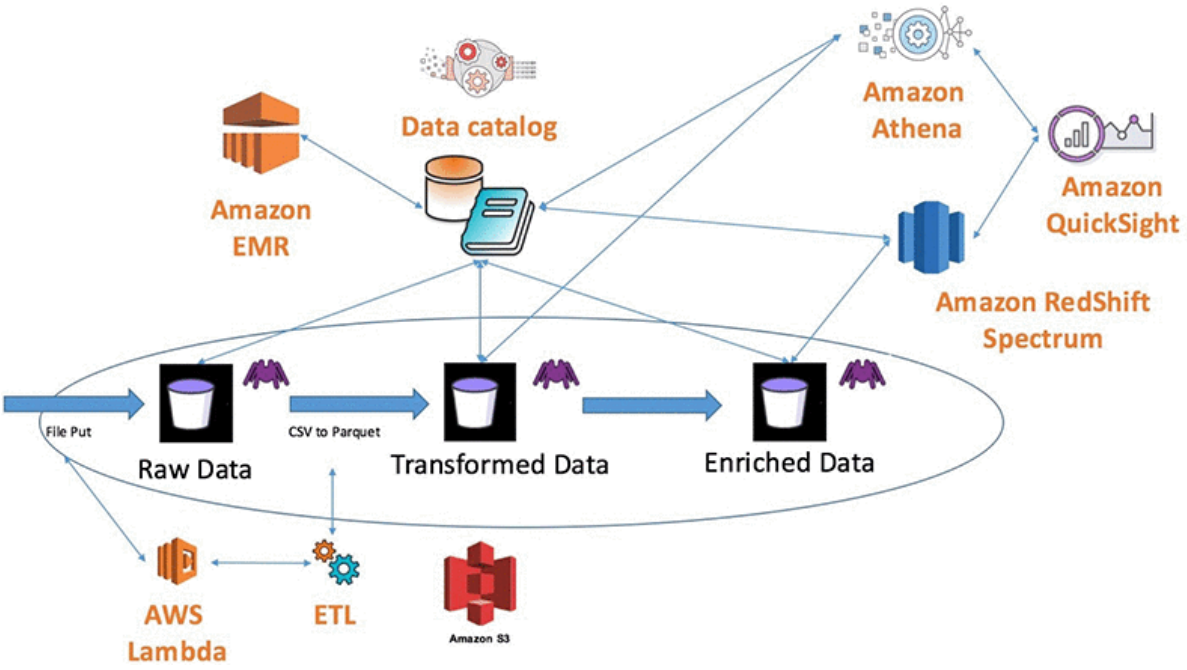

Figura 3: Arquitetura de *Data Lake* utilizada

Isto posto, por motivos organizacionais, um banco de dados com o nome de “eedb_006” foi criado no AWS Glue Catalog, assim como foi criado um *Bucket* de nome “eedb-006-2024-2-grupo-2” no AWS S3 para conter todos os dados utilizados neste experimento.

Na sequência, no banco de dados supracitado, foram criadas quatro tabelas com o nome de “inep2022” sucedidas pelo caractere “_” (*underscore*) e o tipo de codificação utilizado para armazenar os dados das respectivas tabelas, quais sejam: CSV, JSON, Avro e Parquet.

Além disso, um Jupyter Notebook hospedado pelo AWS EMR foi utilizado para executar o código de programação responsável por ler os dados brutos do *Bucket* que contém os dados do Censo da Educação Superior e carregá-los nas quatro tabelas criadas. Por fim, o AWS Athena foi utilizado para consultar as tabelas, fornecendo informações relativas ao tempo de leitura e espaço de armazenamento das tabelas.

Resultados

Após a criação das quatro tabelas utilizando as codificações JSON, CSV, Avro e Parquet, e realizando uma consulta de todos os dados, obteve-se a tabela abaixo, a qual apresenta o não apenas o tempo de escaneamento dos dados, mas também o espaço de armazenamento utilizado pelas tabelas.

Tabela com dados codificados	Escaneamento completo (segundos)	Espaço de armazenamento (*Megabytes*)
JSON	14.392	2340.00
CSV	13.556	326.17
Avro	10.196	63.90
Parquet	13.855	26.45

Tabela 1: Consulta em tabelas com diferentes codificações de dados

No que tange ao tempo de escaneamento completo dos dados, com a Tabela 1 nota-se que o Avro forneceu um tempo de escaneamento (10.196 segundos) levemente menor, se comparado às demais codificações. No entanto, observando-se o espaço de armazenamento, vê-se uma diferença significativa entre as codificações textuais (JSON e CSV) e as codificações binárias (Avro e Parquet). Mais especificamente, a codificação Parquet apresentou o menor espaço de armazenamento, com 26.45 *Megabytes*; já o JSON, a maior, com 2340 *Megabytes*.

Considerando, logo, que no AWS Athena paga-se proporcionalmente pela quantidade de dados lidos por consulta, evidencia-se que a escolha por codificações binárias neste contexto apresenta-se como uma opção eficiente.

Conclusão

Com o trabalho, pôde-se conhecer diferentes tipos de codificações, textuais e binárias, as quais possuem contextos apropriados para serem utilizadas. No que tange às codificações textuais, tais como JSON, XML e CSV, tem-se um formato inteligível ao ser humano e consensual para com a troca de dados entre instituições e serviços, em detrimento de um maior espaço de armazenamento.

Já no que se refere às codificações binárias, e.g., Avro e Parquet, estas apresentam alta capacidade de economia de armazenamento, além de serem otimizadas para consultas, ao custo de não serem inteligíveis ao ser humano; portanto sendo aplicadas normalmente em soluções internas de instituições, normalmente em *Data Lakes*.

Assim, apesar deste trabalho realçar as diferenças entre os diferentes tipos de codificação, não intentou-se eleger a melhor, nem a pior, codificação, e sim realçar suas vantagens, desvantagens e casos de uso, apoiando-se não apenas na literatura científica vigente, mas também em testes empíricos realizados em ambientes de *Data Lake* controlados.

Bibliografia

1. KLEPPMANN, Martin. **Designing Data-Intensive Applications.** 1 ed. Sebastopol, California: O'Reilly, v. 1, 2017. ISBN: 1449373321. Disponível em: https://learning.oreilly.com/library/view/designing-data-intensive-applications/9781491903063/. Acesso em: 14 jul. 2024.

2. INEP. Ministério da Educação. **Censo da Educação Superior**. Brasil: INEP, 2022. Disponível em: https://www.gov.br/inep/pt-br/acesso-a-informacao/dados-abertos/microdados/censo-da-educacao-superior. Acesso em: 16 jul. 2024.

3. HEINRICH, Gordon. Build a Data Lake Foundation with AWS Glue and Amazon S3. *In:* AWS. **AWS Big Data Blog.** Estados Unidos, 27 out. 2017. Disponível em: https://aws.amazon.com/pt/blogs/big-data/build-a-data-lake-foundation-with-aws-glue-and-amazon-s3/. Acesso em: 16 jul. 2024.

4. DATABRICS. **Apache Parquet**. Estados Unidos: Databrics, 2024. Disponível em: https://parquet.apache.org/. Acesso em: 19 jul. 2024.

5. **The Parquet Format and Performance Optimization Opportunities**. Estados Unidos, Boudewijn Braams, 2019. 1 vídeo (40:45). Publicado pela Databricks. Disponível em: https://www.youtube.com/watch?v=1j8SdS7s_NY. Acesso em: 19 jul. 2024.