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 contextualização 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.

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.

Por fim, o grupo motivou-se a realizar testes empíricos com diferentes tipos de codificação, textuais e binárias, a fim de elucidar de modo concreto as diferenças entre codificações para o armazenamento de dados. Para acesso integral a estes testes, acesse o repositório Github disponível em mdspena/eEDB-006-2024-2.

Contextualização

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, coexistirão.

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; 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 serializar 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 linguagens de programação e plataformas, 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 esquema, 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 representar binários com 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.

A seguir, tem-se a representação de um usuário e seus dados, os quais foram codificados em JSON, consumindo 81 *bytes* para serem armazenados.

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

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.

Dentre as várias codificações binárias, incluindo às disponíveis para JSON e XML, destaca-se o Avro. Seu surgimento em 2009 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):

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

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.

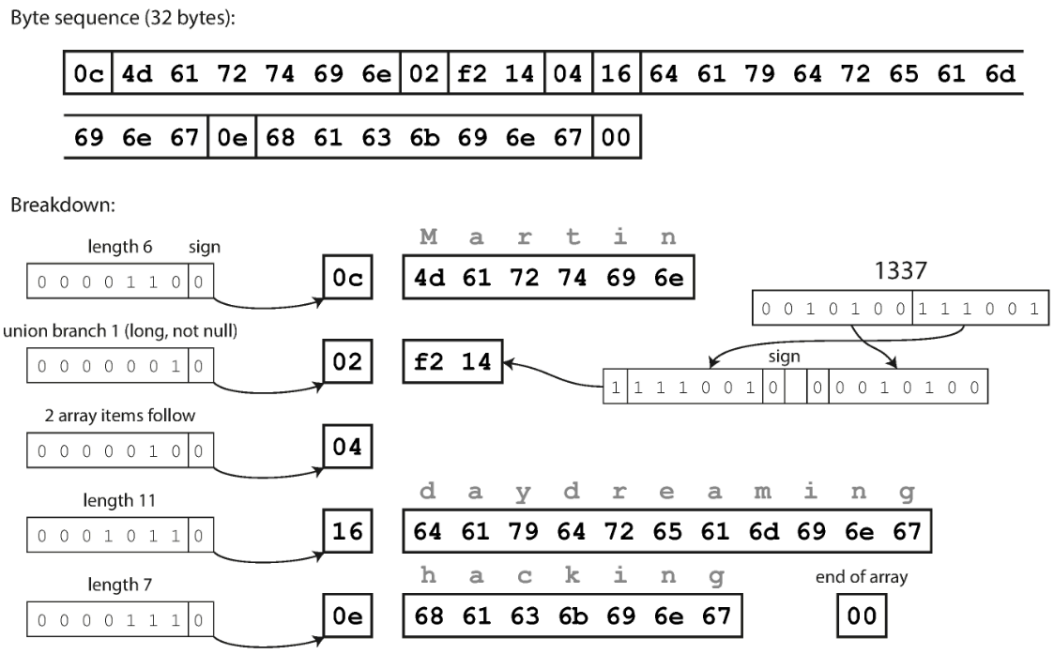

Figura 1: detalhamento sobre funcionamento da codificação binária Apache AVRO

Nesse contexto, 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.

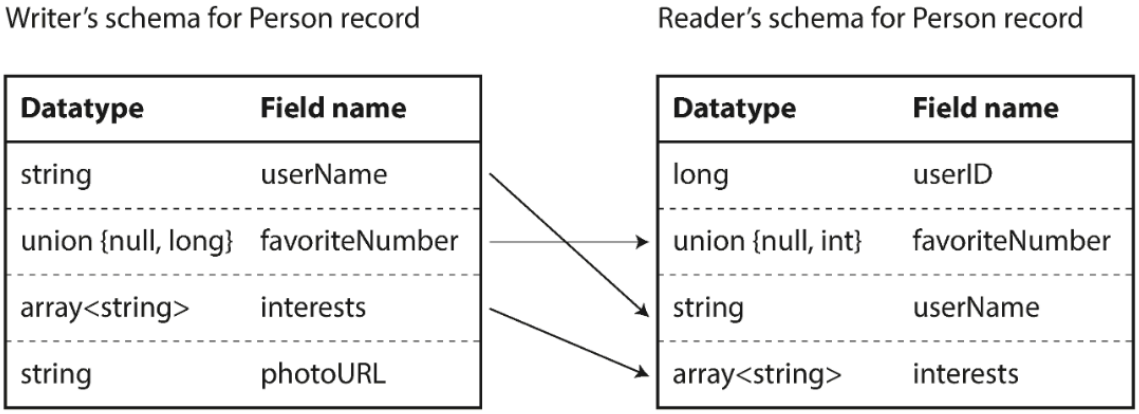

Figura 2: comparação entre esquema de escrita e de leitura para Apache AVRO

Outra vantagem da adoção de esquemas 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.

Testes Empíricos

Com o intuito de realizar testes empíricos e delimitar concretamente as diferenças entre as codificações de dados apresentados neste trabalho, 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.

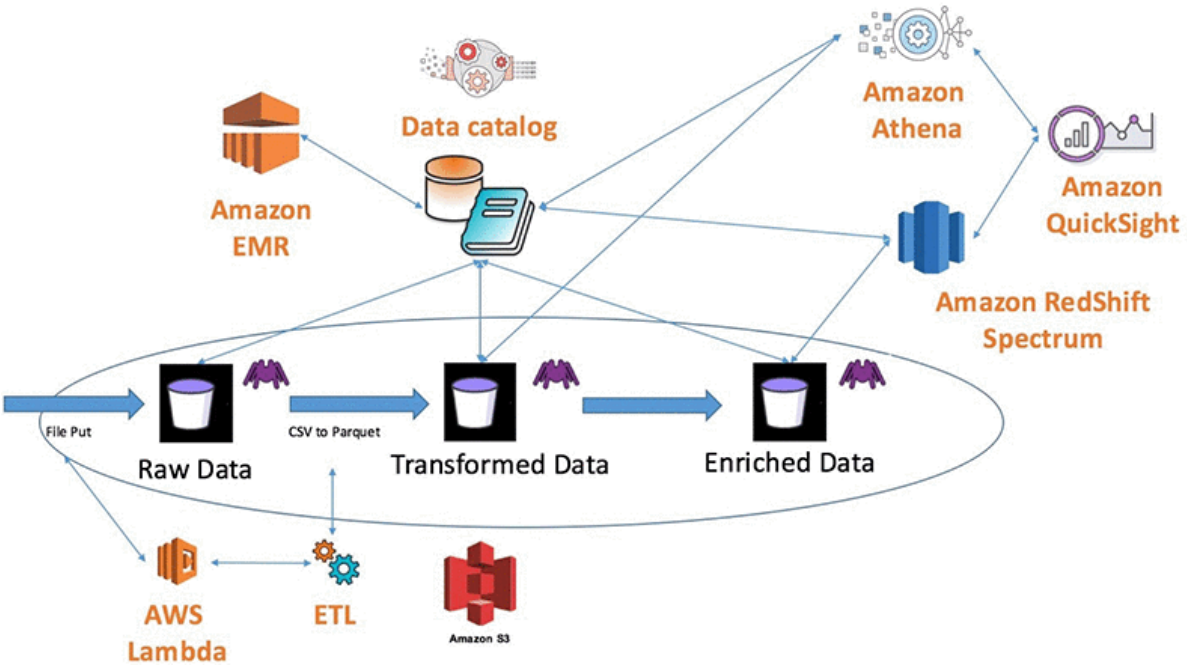

Figura 3: Arquitetura de *Data Lake* utilizada

Isto posto, por motivos organizacionais, um banco de dados com o nome de “eedb_006” foi criado no AWS Glue Catalog, assim como 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 desta tabela, quais sejam: CSV, JSON, Parquet e AVRO.

Além disso, um Jupyter Notebook hospedado pelo AWS EMR foi utilizado para executar código de programação responsável por ler os dados brutos do bucket S3 quanto ao Censo da Educação Superior e carregá-los nas quatro tabelas criadas.

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. 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.