Otimizações e Engenharia de dados com Spark

"Um profissional dessa área é responsável por criar pipelines de dados, manter e construir sistemas de armazenamento de dados que serão utilizados por outras áreas da empresa." (<https://www.letscode.com.br/blog/qual-a-diferenca-de-analista-cientista-e-engenheiro-de-dados>)

## Propósito

Tratando-se de uma ferramenta amplamente utilizada em ambiente de Big Data, o Apache Spark apresenta diferentes ferramentas para que sejamos capazes de aproveitar ao máximo o poder computacional de nosso cluster ao lidar com grande volumetria de dados. Dessa forma, este material visa compartilhar uma série de indicações para melhor usufruto do framework.

## Gerenciamento Delta Lake

### Acessar histórico de tabelas

A combinação Apache Hive e Apache Spark é bastante eficiente para processar e organizar dados, atribuindo aspecto tabular a arquivos e diretórios. Com o surgimento de tecnologias como o Delta Lake, e a arquitetura lakehouse, esta estrutura adquire ainda mais robustez permitindo transações ACID, e consequentemente possibilidades de navegar por diferentes versões, realizar rollbacks e restores.

#### Histórico - Time travel (viagem no tempo)

Como todas as alterações em tabelas do Delta Lake são armazenadas em um log de transações, é possível revisar seu histórico. Através das funções DESCRIBE HISTORY é possível verificar todas as versões de uma tabela:

SQL:

DESCRIBE HISTORY '/path/to/table/' -- retorna todo histórico da tabela

DESCRIBE HISTORY delta.`/path/to/table/`

DESCRIBE HISTORY '/path/to/table/' LIMIT 1 -- retorna apenas a última operação

DESCRIBE HISTORY delta\_table

PySpark:

from delta.tables import \*

deltaTable = DeltaTable.forPath(spark, pathToTable)

fullHistoryDF = deltaTable.history() # retorna todo histórico da tabela

lastOperationDF = deltaTable.history(2) # retorna apenas a última operação

Caso deseje consultar os dados ou status daquela tabela em uma determinada versão, é possível através dos comandos:

SQL:

SELECT \* FROM delta\_table VERSION AS OF 2 -- O número indica a versão

SELECT \* FROM delta\_table@v2 -- O número indica a versão

SELECT \* FROM delta.`/path/to/table@v2` -- O número indica a versão

PySpark:

df = spark.read \

.format("delta") \

.option("versionAsOf", "2") \ # O número indica a versão

.load("/path/to/table")

#### Restore

A fim de restaurar tabelas para versões anteriores, é possível utilizar os comandos:

SQL:

RESTORE TABLE delta\_table TO VERSION AS OF 2 -- O número indica a versão

RESTORE TABLE delta\_table TO TIMESTAMP AS OF current\_timestamp() - INTERVAL '1' HOUR; -- Restaura a tabela para seu status de uma hora atrás.

PySpark:

from delta.tables import \*

deltaTable = DeltaTable.forPath(spark, "/tmp/delta-table")

deltaTable.restoreToVersion(1)

### Vacuum

Com o versionamento de tabelas delta, e a possibilidade de viajar no tempo (time travel), arquivos de todas as versões são mantidos, de forma a tornar este conjunto de dados muito grande e custoso.

A fim de limpar versões históricas, pode-se executar os seguintes comandos:

SQL:

VACUUM students RETAIN 0 HOURS -- desta forma todas as versões anteriores serão apagadas

VACUUM students RETAIN 0 HOURS DRY RUN -- com DRY RUN poderá ver uma lista de arquivos a serem deletados

PySpark:

VACUUM eventsTable -- irá reter apenas arquivos de versões com tempo de existência dentro do padrão definido

VACUUM '/path/to/table' -- limpa tabelas a partir de seu diretório

VACUUM delta.`/path/to/table/`

VACUUM delta.`/path/to/table/` RETAIN 100 HOURS -- irá reter arquivos de versões com menos de 100 horas de existência

VACUUM eventsTable DRY RUN -- com DRY RUN poderá ver uma lista de arquivos a serem deletados

Por padrão o comando Vacuum o previne de deletar arquivos com menos de sete dias de existência. Caso queira alterar, é possível através da seguinte configuração do Spark:

SET spark.databricks.delta.retentionDurationCheck.enabled = false;

É possível também que a execução de Vacuum não esteja habilitada, é possível corrigir através da configuração:

SET spark.databricks.delta.vacuum.logging.enabled = true;

### Optimize e Z-Order

Por diferentes razões, muitas vezes relacionadas a ingestão, não é incomum que conjuntos de dados estejam distribuídos em muitos arquivos pequenos, o que é bastante prejudicial para a performance do Apache Hive. Desta forma, através da função OPTIMIZE é possível reorganizar arquivos em tamanho pré-definido (o padrão no Databricks é de 1 Gb). Veja os diferentes exemplos abaixo em SQL e Python onde a função é aplicada referenciando o caminho da tabela, ou seu nome nos metadados do Hive.

SQL:

OPTIMIZE '/path/to/delta/table' -- optimizes the path-based Delta Lake table

OPTIMIZE delta\_table\_name;

OPTIMIZE delta.`/path/to/delta/table`;

OPTIMIZE delta\_table\_name WHERE date >= '2023-01-31'

PySpark:

from delta.tables import \*

deltaTable = DeltaTable.forPath(spark, pathToTable) # através do caminho

# deltaTable = DeltaTable.forName(spark, tableName) # para tabelas com metadados no Hive

deltaTable.optimize().executeCompaction()

deltaTable.optimize().where("date='2023-01-31'").executeCompaction()

Note que também é possível utilizar a cláusula WHERE em ambas as linguagens para especificar um subconjunto ou partição a ser otimizada. O tamanho padrão dos arquivos pode ser alterado através da configuração:

spark.databricks.delta.optimize.maxFileSize

## Inserção incremental

Realizar cargas incrementais podem ser tarefas desafiadoras para engenheiros de dados. Caso o schema dos dados seja consistente e registros duplicado sejam tratados previamente, a função COPY INTO propiciará uma forma idempotente de carregar dados incrementalmente de sistemas externos, de forma mais eficiente que através de full scans (lentos e custosos por analisar todos os dados de uma tabela).

SQL:

COPY INTO delta\_table

FROM "path\_to\_table"

FILEFORMAT = PARQUET

## Clonagem

Um Delta Lake apresenta diferentes opções para copiar tabelas de forma eficiente, dentre elas destaca-se o comando CLONE e suas variações:

### Deep Clone

DEEP CLONE copia todos dados e metadados de uma tabela origem para uma tabela destino. Esta cópia acontece de forma incremental, de forma que ao executar o mesmo comando novamente, apenas alterações são atualizadas.

SQL:

CREATE OR REPLACE TABLE delta\_table\_clone

DEEP CLONE delta\_table

Esta função tem uma aplicabilidade interessantíssima para DR (Disaster Recovery - Recuperação de Disastre) e pode ser consultada na seção de bibliografia e indicações deste material.

### Shallow Clone

Como DEEP CLONE copia todos os dados de uma tabela, em grandes conjuntos de dados sua execução pode levar bastante tempo. Caso precise apenas de uma cópia rápida apenas para aplicar ou testar mudanças, sem pôr a tabela original em risco, SHALLOW CLONE pode ser uma boa opção. Através dele, apenas logs de transação (recurso e característica fundamental de tabelas Delta) são copiados, não os dados.

SQL:

CREATE OR REPLACE TABLE delta\_table\_clone

DEEP CLONE delta\_table

## Views

Views são conceitos oriundos de bancos relacionais que se baseiam em resultados de queries, criando tabelas virtuais onde seus dados não são persistidos.

Em Spark o mesmo conceito existe e se aplica de forma similar, entretanto, através dos parâmetros GLOBAL e TEMPORARY, podendo ter sua durabilidade e acesso alterados.

SQL:

CREATE [[GLOBAL] TEMPORARY] VIEW nome\_view

SELECT \* FROM nome\_tabela

Uma View do tipo temporária tem sua existência vinculada a sessão Spark, que quando finalizada também extingue a tabela. Já as views temporárias globais se diferem por estarem ao Cluster Spark, e acessíveis através de um banco de dados temporário chamado global\_temp.

## Cashe

Ainda que o Apache Spark ofereça processamento significativamente mais rápido que outras formas de processamento distribuído como o MapReduce, é possível tornar a manipulação de DataFrames ainda mais ágil através do Cache de um cluster.

É importante citar que esta prática é mais comum na rotina de cientistas e analistas de dados do que de engenheiros, visto que ainda que esta memória seja mais rápida, é mais limitada em volume e disponibilidade, de forma a prejudicar o desempenho do cluster caso conjuntos de dados muito grandes sejam armazenados ali, ou mesmo por muito tempo.

PySpark:

dataframe.spark.cache()

Para remover um dataframe do cache, utilize o método unpersist:

PySpark:

df.spark.unpersist()

Através do método persist é possível indicar o "Storage Level", ou nível de armazenamento, para o cashing de um Data Frame. Ao utilizar o método cache visto anteriormente, o resultado gerado é o mesmo da execução de:

PySpark:

df.spark.persist(StorageLevel.MEMORY\_AND\_DISK)

Onde o nível de armazenamento é "MEMORY\_AND\_DISK". Outras opções são MEMORY\_ONLY, MEMORY\_AND\_DISK, MEMORY\_ONLY\_SER, MEMORY\_AND\_DISK\_SER, DISK\_ONLY, MEMORY\_ONLY\_2, MEMORY\_AND\_DISK\_2. Para mais detalhes consulte a seção de indicações e bibliografia deste material, e a documentação oficial do Apache Spark.

## Indicações e bibliografia

[Otimizações](https://docs.delta.io/latest/optimizations-oss.html#language-python)

[Otimizar - Delta Lake](https://docs.databricks.com/sql/language-manual/delta-optimize.html)

[Delta utility](https://docs.delta.io/latest/delta-utility.html)

[Versionamento - Delta Lake](https://docs.databricks.com/delta/history.html)

[Restuarar versões](https://docs.databricks.com/sql/language-manual/delta-restore.html)

[Restaurar versões](https://delta.io/blog/2022-10-03-rollback-delta-lake-restore/)

[Vacuum - Delta Lake](https://docs.databricks.com/sql/language-manual/delta-vacuum.html)

[Disaster Recovery Delta Lake](https://www.databricks.com/wp-content/uploads/notebooks/using-deep-clone-disaster-recovery-delta-lake-databricks.html)

[Copy Into - Delta Lake](https://docs.databricks.com/sql/language-manual/delta-copy-into.html)

[Pandas API Spark](https://spark.apache.org/docs/3.2.0/api/python/reference/pyspark.pandas/api/pyspark.pandas.DataFrame.spark.persist.html)