Filipe de Melo Silva

AVALIAÇÃO DA UTILIZAÇÃO DE ARQUIVOS DESNORMALIZADOS NO SPARK SQL

Universidade Federal de Pernambuco
graduacao@cin.ufpe.br
www.cin.ufpe.br/~graduacao

Recife
2021

Filipe de Melo Silva

AVALIAÇÃO DA UTILIZAÇÃO DE ARQUIVOS DESNORMALIZADOS NO SPARK SQL

Trabalho apresentado ao Programa de Graduação em Engenharia de Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Bacharel em Engenharia da Computação.

Orientador: *Robson do Nascimento Fidalgo*

Recife
2021

RESUMO

A constante redução no custo de armazenamento tem influenciado as organizações a coletarem mais e mais dados. Nesse cenário, o uso de banco de dados relacionais pode ser inviável para realizar análises por, muitas vezes, serem limitados à escalabilidade vertical. Por conta disso, diversas ferramentas com a capacidade de processar dados de forma distribuída vêm sendo desenvolvidas. Uma das mais populares é o Spark, graças ao seu suporte a SQL e sua performance superior ao MapReduce. Entretanto, por conta da distribuição dos dados, operações de junção podem ser custosas. Uma prática largamente utilizada em bancos relacionais para reduzir o uso de junções é a desnormalização dos dados. Esse trabalho avalia o uso da desnormalização nos arquivos de dados utilizados pelo Spark SQL. Para isso, a avaliação será feita utilizando os formatos CSV e Parquet e os dados serão gerados pela ferramenta Star Schema Benchmark. Os resultados mostraram que o CSV, quando desnormalizado aumentou a duração da execução em mais de 3,5 vezes. Já o Parquet, foi equivalente em alguns casos, porém em outros, executou 3 vezes mais rápido.

Palavras-chave: Spark. Star Schema Benchmark. Desnormalização. SSB. Star Schema Benchmark.

ABSTRACT

The constant reduction in the cost of data storage has influenced organizations to collect more and more data. In this scenario, the use of relational databases can be impractical to perform analyzes since they are often limited to vertical scalability. Because of this, several tools with the ability to process data in a distributed way have been developed. One of the most popular is Spark, thanks to its SQL support and its superior performance over MapReduce. However, because of data distribution, join operations can be expensive. A widely used practice in relational databases to reduce the use of joins is data denormalization. This work evaluates the use of denormalization in the data files used by Spark SQL. For this, the evaluation will be done using CSV and Parquet formats and the data will be generated by the Star Schema Benchmark tool. In the proposed environment, the CSV and Parquet format had different results. Denormalized CSV increased the execution time by more than 3.5 times. Parquet, on the other hand, was equivalent in some cases, but in others, it ran 3 times faster.

Keywords: Spark. Star Schema Benchmark. Denormalization. SSB. Star Schema Benchmark.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1

INTRODUÇÃO

O uso de dados tem se tornado cada vez mais presente nas corporações. Esse processo é influenciado por diversos fatores como o aumento da competitividade do mercado e a constante redução do custo de armazenamento dos dados (CHAMBERS; ZAHARIA, 2018). Esses dados são utilizados para diversos fins como gerar insights ou identificar problemas antecipadamente.

Visando tratar esse problema de forma centralizada, o *data warehouse* (DW) tem se tornado cada vez mais popular. Este é um tipo de sistema de gerenciamento de dados projetado exclusivamente para realizar consultas e análises avançadas. Dessa forma, um DW centraliza e consolida grandes quantidades de dados históricos, geralmente originados de diversas fontes como arquivos de armazenamento, bancos de dados e até mesmo *logs* de aplicações.

Por conta da limitação de bancos de dados relacionais em analisar o grande volume de dados presentes no DW, diversas ferramentas vêm sendo desenvolvidas com foco em processamento em cluster. Uma das mais populares é o *framework* Spark, graças ao seu suporte a SQL e sua performance superior ao MapReduce (SILVA; ALMEIDA; QUEIROZ, 2016). Esse *framework* é capaz de processar dados de forma paralelizada através da distribuição desses dados pelo cluster.

Entretanto, operações de junção (JOIN) muitas vezes são ineficientes em cenários onde os dados estão distribuídos. Isso acontece por conta da necessidade de acesso à rede para obtenção de dados durante essa operação, uma vez que nem todos os dados a serem utilizados estão presentes em uma mesma máquina. Diversos estudos tentam otimizar esse tipo de operação (ZHANG et al., 2016; CHENG et al., 2020; DAVIDSON, 2013). Uma prática largamente utilizada em bancos relacionais para reduzir o uso de junções é a desnormalização dos dados. Nessa prática, dados redundantes são adicionados nas linhas de uma tabela afim de evitar a necessidade de cruzamento com outras tabelas.

Esse trabalho tem como objetivo avaliar o impacto da desnormalização de arquivos quando usados como entrada no Spark SQL. Para isso, a avaliação será feita utilizando os formatos CSV e Parquet.

2

CONCEITOS

2.1 STAR SCHEMA BENCHMARK

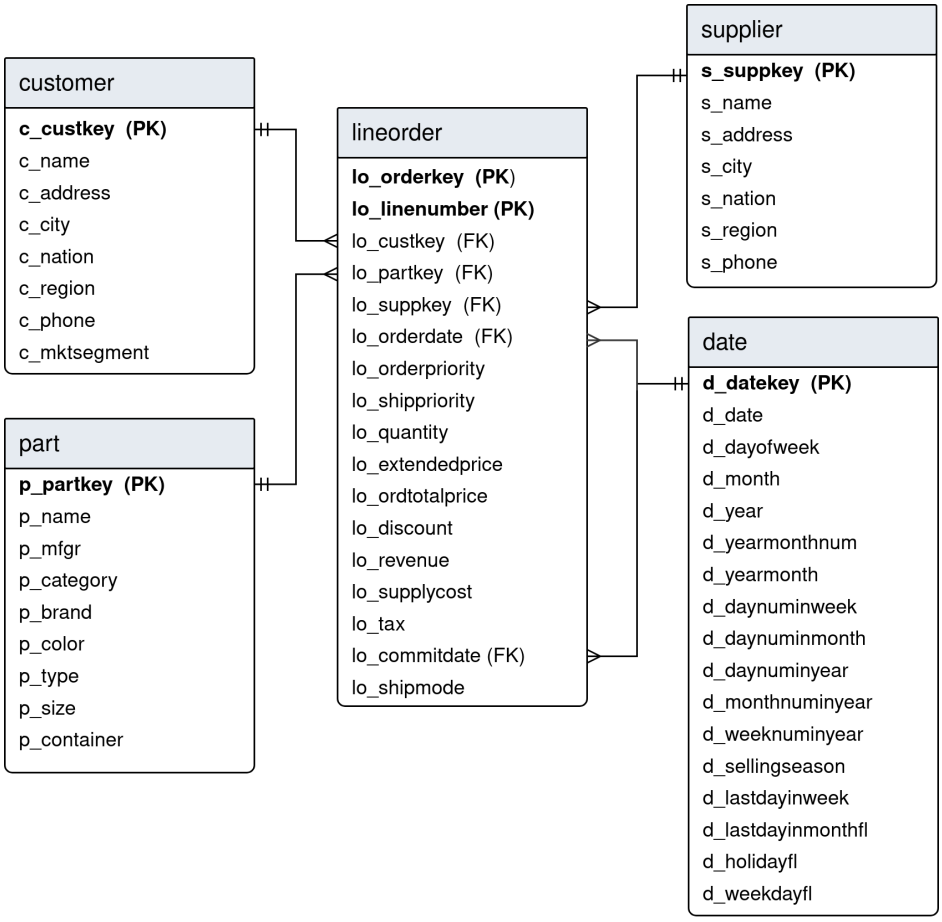

Figura 1: Tabelas do SSB

O Star Schema Benchmark (SSB) (O'NEIL; O'NEIL; CHEN, 2009) é uma ferramenta de benchmark que serve para medir a performance de bancos de dados utilizados em cenários de *Data Warehouse*. Ele foi criado a partir da transformação do esquema do TPC-H (POESS; FLOYD, 2000) para um esquema estrela.

```
1|Supplier#000000001|sdrGnXCDRcfriBvY0KL,i|...|27-989-741-2988|
2|Supplier#000000002|TRMhVHz3XiFu          |...|15-768-687-3665|
3|Supplier#000000003|BZ0kXcHUcHjx62L7CjZS |...|11-719-748-3364|
```

Figura 2: Primeiras linhas do arquivo supplier.tbl gerado pelo dbgen com fator de escala 1.

Tabela 1: Numero de linhas e colunas de cada tabela do SSB. A quantidade aproximada de linhas, com exceção da tabela Part se dá em função do fator de escala.

Nome da Tabela	Número Aproximado de Linhas	Número de Colunas
Customer	$30.000 * SF$	8
Date	2.556	17
Part	$2.000 * SF$	9
Supplier	$200.000 * [1 + log_2 SF]$	7
LineOrder	$6.000.000 * SF$	17

Através de um programa chamado dbgen, o SSB gera dados históricos referentes a pedidos de uma empresa de varejo, durante um certo período de tempo. Esse dados são estruturados utilizando as quatro tabelas de dimensão Customer, Supplier, Part e Date, e a tabela de fatos LineOrder, mostradas na Figura 1. Para cada uma das cinco tabelas é criado um arquivo tbl distinto com as colunas separadas por uma barra vertical (‘|‘), como mostra a Figura 2. Além disso, a quantidade de linhas pode ser controlada através do parâmetro fator de escala (SF), cujo valor pode variar de 1 até 1000. A Tabela 1 apresenta a quantidade de linhas em função do fator de escala.

Tabela 2: Tabelas de dimensão utilizadas em cada classe de consulta proposta no SSB.

Consulta	Dimensões
Q1	Date
Q2	Date, Part, Supplier
Q3	Date, Customer, Supplier
Q4	Date, Customer, Supplier, Part

Além de fornecer os dados, o SSB também define 13 consultas, onde algumas foram adaptadas do TPC-H e outras criadas para validar o novo esquema. Essas consultas são divididas em quatro classes (Q1, Q2, Q3, Q4), onde cada uma testa diferentes variações de seletividade e de agregação. Todas as consultas utilizam a tabela LineOrder e as tabelas de dimensão utilizadas em seus relacionamentos são apresentadas na Tabela 2.

2.2 SPARK

O Apache Spark é um *framework* criado para facilitar processamento em paralelo em um *cluster* (BRAAMS, 2018). Esse *framework* pode ser utilizado em vários linguagens como Java, Scala, Python. A principal característica do Spark é carregar os dados em memória, e só então realizar as consultas. Por conta disso, o Spark possui uma ótima performance, sendo esse mil vezes mais rápido que o MapReduce do Hadoop (SILVA; ALMEIDA; QUEIROZ, 2016).

Para funcionar em *cluster* existem diversos gerenciadores disponíveis como *Standalone Cluster*, YARN e Spark Mesos. Cada um deles possui diferentes níveis de segurança e disponbilidade. Dentre esses, o mais simples de utilizar é o *Standalone Cluster*, pois já vem embutido no Spark e pode ser facilmente configurado.

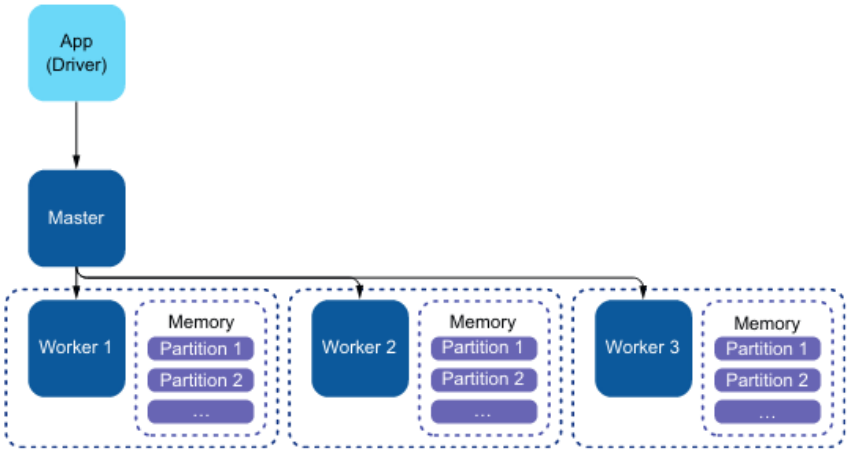

Figura 3: Representação lógica da organização do *Master*, *Worker* e *Driver* no *cluster*. [Retirado de (PERRIN, 2020)].

O *Standalone Cluster* consiste em um processo *master* e alguns processos *workers* executados em máquinas distintas. O processo *master* atua como o gerenciador do *cluster*. Ele reserva nos *workers* os recursos solicitados por uma aplicação, também chamada de *driver*. Já os *workers* são quem propriamente executam as aplicações, e durante essa execução são chamados de *executors*. A Figura 3 ilustra o processo de gerenciamento do Spark.

Além disso, o Spark conta com diversos modulos. Um dos principais é o Spark SQL, usado para realizar processamento de dados estruturados em estruturas chamadas de DataFrames que organiza os dados em colunas nomeadas. Essa estrutura seria equivalente a uma tabela em um banco relacional. Esse módulo também permite o uso da linguagem SQL, utilizando o DataFrame como uma tabela.

O Spark utiliza uma estrutura interna imutável, também chamada de RDD. Dessa forma, toda estrutura é criada a partir de uma fonte externa, como um arquivo, ou a partir da transformação de uma RDD já existente. Essas transformações possuem custos computacionais diferentes. Algumas como o filtro, por exemplo, são mais rápidas, pois conseguem ser realizadas apenas com os dados presentes no próprio *worker*. Outras, como a junção, precisam interagir com outros

workers através da rede, e consequentemente são mais lentas. Esse processo de requisitar dados que não estão presentes localmente é chamado de *shuffle* e existem diversos trabalhos que tentam otimizá-lo (ZHANG et al., 2016; CHENG et al., 2020; DAVIDSON, 2013).

```
== Physical Plan ==
* HashAggregate (14)
+- Exchange (13)
   +- * HashAggregate (12)
      +- * Project (11)
         +- * BroadcastHashJoin Inner BuildRight (10)
            :- * Project (4)
            :  +- * Filter (3)
            :     +- * ColumnarToRow (2)
            :        +- Scan parquet  (1)
            +- BroadcastExchange (9)
               +- * Project (8)
                  +- * Filter (7)
                     +- * ColumnarToRow (6)
                        +- Scan parquet  (5)
...
```

Figura 4: Exemplo de *physical plan* do Spark

Outro detalhe importante é que o Spark utiliza *Lazy Evaluation*, dessa forma, as transformações solicitadas só são realmente executadas quando necessárias, como por exemplo, ao salvar os dados em um arquivo ou exibi-los. Com isso, o Spark pode realizar otimizações na sequência de transformações e criar uma nova sequência de instruções (*physical plan*) otimizadas para serem executado no *cluster*. O *physical plan* consiste em um conjunto de instruções em um nível mais baixo, como pode ser visto na Figura 4. Nele, por exemplo, é determinado qual será o algoritmo utilizado numa operação de junção, que será visto na Subseção 2.2.1.

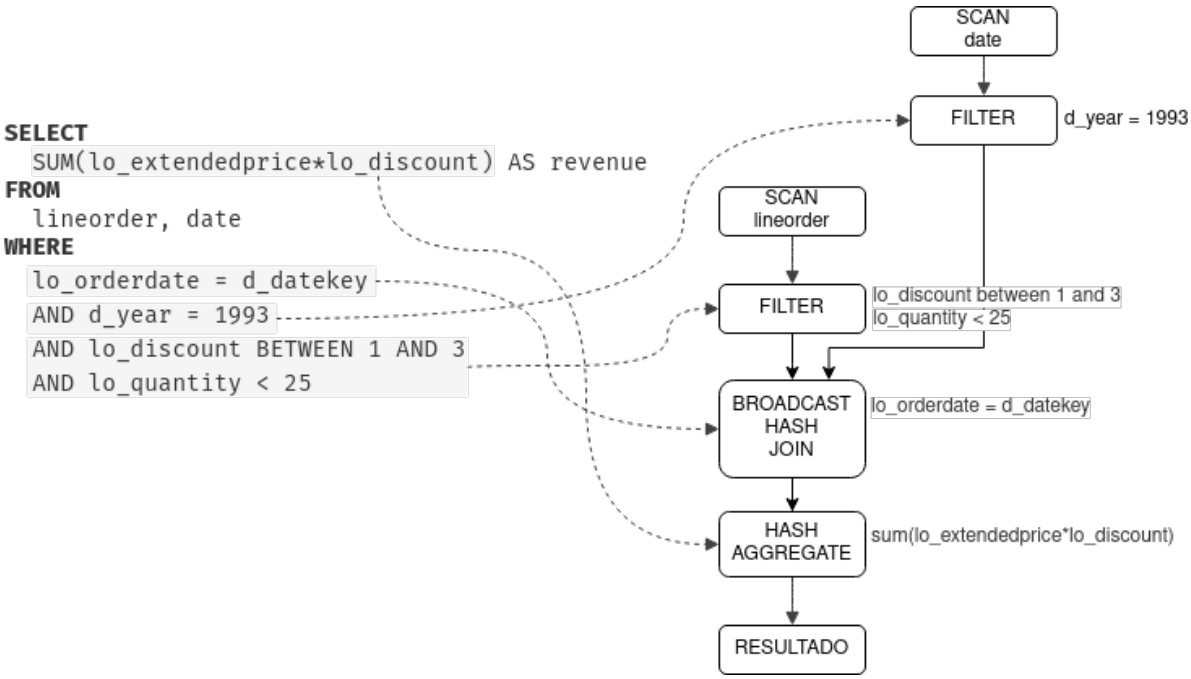

Figura 5: Versão simplificada do *physical plan* extraído do Spark UI. Essa figura representa de forma gráfica, o *physical plan* apresentado na Figura 4

Para visualizar esse *physical plan*, o Spark possui uma *interface web* chamada de Spark UI. Além dessa informação, é possível verificar outras como o volume de dados transferidos em operações de *shuffle*. O lado direito da Figura 5 mostra uma simplificação da versão gráfica obtida no Spark UI, e o lado esquerdo apresenta a consulta que gerou essa visualização.

Além do Spark UI, existem outras ferramentas de monitoramento que podem ser utilizadas para coletar mais informações do Spark. Uma delas é o Netdata, uma ferramenta Open Source criada para coletar métricas de uma máquina em tempo real. Essa ferramenta consegue obter resultados sobre CPU, disco, memória, entre outras coisas. Além disso, é possível coletar informações sobre os processos do Spark.

2.2.1 Join

O Apache Spark possui diversos algoritmos para realizar a junção de tabelas. Quando a condição de junção é a igualdade entre duas colunas ordenaveis, existem dois tipos principais: *Broadcast Hash Join*(BHJ) e *Sort Merge Join*(SMJ) (BRAAMS, 2018).

Na configuração padrão, o BHJ é aplicado apenas quando uma das tabelas utilizadas na junção possui um tamanho inferior a 10MB. Nesse caso, uma *hash table* é construída tendo como chave o *hashing* das colunas de junção da tabela menor, e como valor, as colunas dessa tabela que serão utilizadas na consulta. Após a criação, a *hash table* é transferida e utilizada em todos os executores que fazem a varredura dos dados da tabela maior.

Quando um BHJ não é possível, um SMJ é utilizado. Essa operação funciona primeiro ordenando ambas as tabelas pelas colunas utilizadas na junção e, em seguida, aplicando uma

varredura intercalada sobre os dois resultados. Dessa forma, é possível diminuir o número de comparações entre os dados das tabelas, reduzindo assim o *shuffle*.

Um BHJ é preferível a um SMJ, pois é significativamente mais rápido em muitas situações. Isso se dá pelo fato de o BHJ, não requerer qualquer *shuffle* de dados pela rede durante a execução, exceto para a transmissão da tabela *hash*. Além disso, pesquisas em *hash table* são realizadas em tempo constante. Um SMJ, no entando, requer que ambas as tabelas sejam ordenadas, o que pode ser uma operação custosa. No contexto do Spark, isso pode ser ainda mais custoso, já que os dados são geralmente distribuídos e, portanto, existe a necessidade de *shuffle* pela rede.

2.2.2 Formatos de arquivo

O Spark possui suporte a diversos formatos de arquivo e dentre os mais usados está o CSV e o Parquet. O CSV (*Comma Separated Values*) é arquivo de texto plano suportado por diversas ferramentas como o Microsft Excel e GoogleSheets. Nesse formato, cada linha representa um registro, e as colunas são separadas por um delimitador (normalmente uma vírgula). Já o formato Parquet é um formato de armazenamento colunar desenvolvido com foco em grandes cargas de trabalho.

Ler um arquivo Parquet no Spark é mais eficiente que ler um arquivo CSV (BELOV; TATARINTSEV; NIKULCHEV, 2021). Uma das razões para isso é o fato de o arquivo Parquet possuir um tamanho reduzido em relação ao arquivo CSV. Outra razão é que no Parquet, as colunas são armazenadas juntas. Por conta disso, o Spark consegue ignorar as colunas que ele não precisa para uma consulta durante a leitura.

3

EXPERIMENTOS

O experimento consiste na avaliação do impacto da desnormalização dos arquivos CSV e Parquet utilizados como entrada no Spark. Para isso, uma aplicação escrita em Python foi submetida ao *cluster* e executou as consultas Q1.1, Q2.1, Q3.1 e Q4.1 do SSB, descritas na Seção 2.1, no esquema normalizado (*star*) e desnormalizado (*flat*). Além disso, essa aplicação coletou as seguintes métricas ao longo de 30 execuções: duração de cada consulta, uso de rede através de operações *shuffle* e uso de memória.

Esse experimento seguirá as quatro etapas essenciais na realização de benchmarks (BELOV CIFERRI, 1995). São elas: (i) esquema e carga de trabalho; (ii) geração de dados; (iii) métricas e parâmetros; e (iv) validação. As etapas (i) e (ii) serão descritas na Seção 3.1, a etapa (iii) na Seção 3.2 e por fim, a etapa (iv) será descrita na seção Seção 3.3.

3.1 CRIAÇÃO DA ENTRADA

Tabela 3: Quantidade de linhas obtidas para cada fator de escala (SF).

Tabela	$SF = 1$	$SF = 10$	$SF = 100$
Customer	30.000	300.000	3.000.000
Date	2.556	2.556	2.556
Part	200.000	800.000	1.400.000
Supplier	2.000	20.000	200.000
LineOrder	6.001.171	59.986.214	600.038.145

Uma parte dos dados utilizados no experimento foi obtida através do programa dbgen (gerador de tabelas do SSB). Nesse programa, os fatores de escala 1, 10 e 100 foram utilizados como parâmetro, gerando arquivos tbl com as quantidades de linhas apresentadas na Tabela 3. É possível perceber que o número de linhas da tabela LineOrder difere do valor esperado pela fórmula apresentada na Tabela 1, uma vez que esta calcula um valor aproximado. Como os formatos Parquet e CSV não podem ser obtidos diretamente do dbgen, os arquivos tbl foram convertidos, resultando em 30 arquivos (15 em cada formato).

```
SELECT
    lineorder.lo_orderkey, lineorder.lo_linenumber, ...,
    date.d_date, date.d_dayofweek, ...,
    part.p_name, part.p_mfgr, ...,
    customer.c_name, customer.c_address, ...,
    supplier.s_name, supplier.s_address, ...,
    cd.d_date AS cd_date, cd.d_dayofweek AS cd_dayofweek, ...,
FROM lineorder
JOIN date ON lineorder.lo_datekey = date.d_datekey
JOIN part ON lineorder.lo_partkey = part.p_partkey
JOIN customer ON lineorder.lo_custkey = customer.c_custkey
JOIN supplier ON lineorder.lo_suppkey = supplier.s_suppkey
JOIN date AS cd ON lineorder.lo_commitdate = cd.d_datekey
ORDER BY lineorder.lo_orderkey, lineorder.lo_linenumber
```

Figura 6: Consulta utilizada para a desnormalização dos dados do SSB.

Tabela 4: Tamanho dos arquivos CSV e Parquet para cada fator de escala (SF).

	$SF = 1$		$SF = 10$		$SF = 100$	
Tabela	**CSV**	**Parquet**	**CSV**	**Parquet**	**CSV**	**Parquet**
Customer	2,7M	1,2M	28M	12M	274M	118M
Date	224K	32K	224K	32K	224K	32K
Part	17M	2,3M	65M	8,8M	115M	16M
Supplier	164K	84K	1,6M	800K	17M	7,8M
LineOrder	562M	153M	5,7G	1,6G	59G	17G
FlatLineOrder	2,7G	313M	27G	3,6G	263G	54G

Outra parte da entrada foi obtida através da desnormalização dos arquivos gerados anteriormente. Utilizando a consulta da Figura 6 no Spark, as tabelas de dimensão Customer, Supplier, Part e Date foram unidas com a tabela de fatos LineOrder. Assim, todos os 65 atributos foram organizados dentro de uma única tabela, aqui chamada de FlatLineOrder. Além disso, foram removidas todas as 11 chaves (primárias e estrangeiras) utilizadas em junções (*JOIN*). Para garantir que as linhas do arquivo desnormalizado mantenham a mesma ordem da entrada, a consulta as ordena pelos mesmos dois índices usados na tabela LineOrder. Ao final da execução 6 novos arquivos foram obtidos (3 em cada formato). O tamanho de todos os arquivos, incluindo os utilizados no esquema estrela, pode ser visto na Tabela 4.

```
SELECT
    SUM(lo_extendedprice*lo_discount) AS revenue
FROM
    lineorder, date
WHERE
    lo_orderdate = d_datekey
    AND d_year = 1993
    AND lo_discount between 1 AND 3
    AND lo_quantity < 25
```

Figura 7: Consulta Q1.1 proposta pelo SSB.

```
SELECT
    SUM(lo_extendedprice*lo_discount) AS revenue
FROM
    flatlineorder
WHERE
    d_year = 1993
    AND lo_discount between 1 AND 3
    AND lo_quantity < 25
```

Figura 8: Versão desnormalizada da Q1.1 proposta pelo SSB

Finalmente, as 4 consultas utilizadas no esquema estrela foram transformadas em consultas equivalentes no modelo desnormalizado. Para isso, todas as tabelas originalmente utilizadas foram substituidas pela nova tabela FlatLineOrder. Além disso, houve a remoção de todas as operações de relacionamento entre tabelas. A Figura 8 mostra um exemplo da transformação realizada na consulta Q1.1 (Figura 7) proposta pelo SSB.

3.2 CONFIGURAÇÃO DA INFRAESTRUTURA

O ambiente para realizar os benchmarks propostos é composto por quatro servidores (também chamados de nós), sendo um *master* e três *workers*. Todos os servidores possuem o sistema operacional Centos 7, conexão de rede de 1GB/s e um disco mecânico SATA de 7200 RPM. O servidor *master* possui um processador Intel Xeon E5410 e 16GB de memória, enquanto os *workers* possuem um processador intel Xeon E3-1230V6 e 48GB de memória RAM. A versão do Spark 3.1.12 foi utilizado no modo *standalone* e foram reservados 8 núcleos e 40GB em cada *worker* e 4 núcleos e 10GB no *master*.

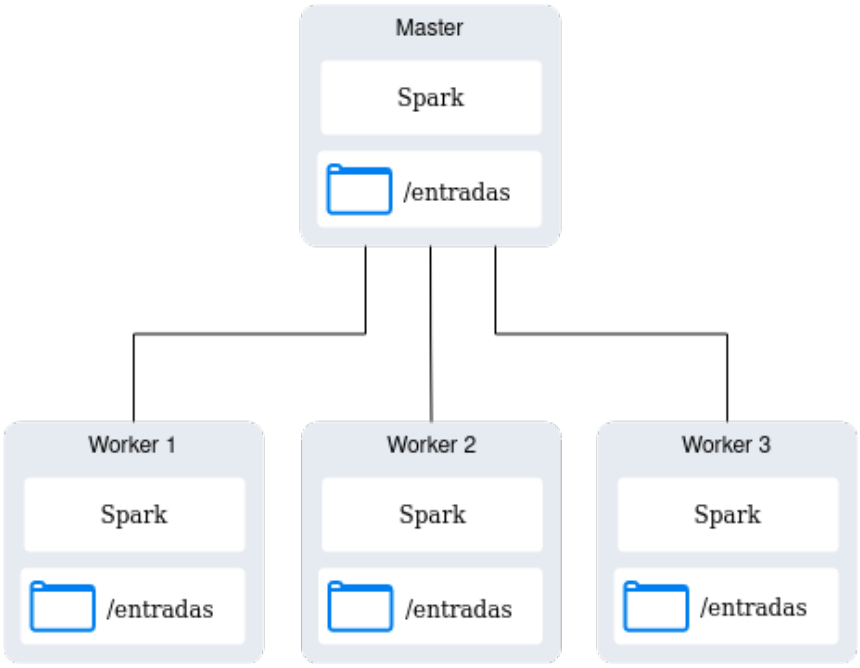

Figura 9: Configuração da entrada nos nós do Spark

Para armazenar os arquivos de entrada, foi utilizado o sistema de arquivo local (ext4) de cada nó. Dessa forma, todos os arquivos (Tabela 4) foram copiados para um mesmo caminho em todos os nós utilizados no Spark, como mostra a Figura 9. Uma consequência desse tipo de armazenamento é o cache em memória realizado pelo sistema operacional após a leitura de um arquivo. Esse recurso pode afetar os testes de performance do experimento, uma vez que reduz o acesso ao disco. Por conta disso, a limpeza do cache de todos os nós foi efetuada entre cada execução.

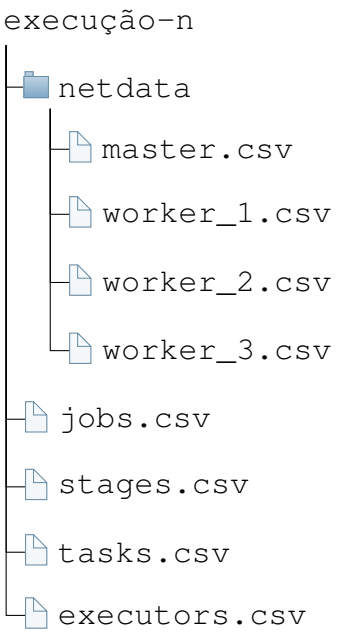

Figura 10: Arquivos gerados pela aplicação em cada execução

Para o monitoramento foi utilizada a *API* do Spark UI e a do Netdata. A primeira foi utilizada para obter informações internas, como por exemplo, o *physical plan*, duração de cada execução e os dados transferidos nas operações de *shuffle*. Já o Netdata foi utilizado para coletar informações sobre processo da JVM utilizada pelo Spark. Todos os dados coletados foram tratados e armazenados em arquivos CSV no nó *master* pela aplicação *driver*, como mostra a Figura 10.

3.3 ANÁLISE DOS RESULTADOS

Tabela 5: Número de *Sort Merge Join* no esquema estrela.
Os *Joins* restantes foram do tipo *Broadcast Hash*.

Entrada	Q1.1	Q2.1	Q3.1	Q4.1	Total
CSV ($SF = 1$)	0	0	0	0	0
CSV ($SF = 10$)	0	1	0	0	1
CSV ($SF = 100$)	0	1	1	1	3
Parquet ($SF = 1$)	0	0	0	0	0
Parquet ($SF = 10$)	0	0	0	0	0
Parquet ($SF = 100$)	0	0	1	1	2

Através da análise do *physical plan* de cada execução, percebeu-se alterações em relação ao tipo de operação de *Join* utilizada. A Tabela 5 descreve a quantidade de operações do tipo SMJ realizadas em cada formato de arquivo e fator de escala. Ao aumentar o fator de escala, o número de operações aumentou em todos os formatos, no entanto, para o Parquet, esse crescimento foi menor. Isso ocorreu pois a decisão de uso do BHJ é baseada no tamanho do arquivo, e os arquivos Parquet possuem tamanho reduzido em relação ao CSV, como pode ser visto na Tabela 4.

Além disso, a consulta Q1.1 não chegou a usar operações de SMJ em nenhuma execução. Isso aconteceu pois o único *Join* utilizado nessa consulta é com a tabela Date, e essa possui um tamanho fixo e dentro do limiar do BHJ. Por conta disso, essa consulta deve permanecer sem operações de SMJ independente do fator de escala.

Tabela 6: Duração em minutos das consultas utilizando o formato CSV. Os valores em negrito representam execuções onde houve operações de *Sort Merge Join*.

Entrada	Q1.1	Q2.1	Q3.1	Q4.1	Total
Star ($SF = 1$)	$4,0 \pm 2,8$	$3,7 \pm 1,0$	$3,1 \pm 0,3$	$3,4 \pm 0,2$	$14,4 \pm 4,4$
Flat ($SF = 1$)	$14,1 \pm 2,1$	$13,7 \pm 1,0$	$13,4 \pm 0,5$	$13,2 \pm 0,5$	$54,5 \pm 3,5$
Star ($SF = 10$)	$27,5 \pm 4,2$	$\mathbf{30,5 \pm 2,0}$	$25,4 \pm 1,1$	$26,2 \pm 1,6$	$109,8 \pm 5,8$
Flat ($SF = 10$)	$109,8 \pm 3,2$	$109,0 \pm 1,9$	$108,9 \pm 2,0$	$108,7 \pm 2,6$	$436,5 \pm 8,3$
Star ($SF = 100$)	$222,0 \pm 5,6$	$\mathbf{226,5 \pm 2,0}$	$\mathbf{225,6 \pm 2,4}$	$\mathbf{229,6 \pm 1,9}$	$903,8 \pm 9,1$
Flat ($SF = 100$)	$1207,9 \pm 7,2$	$1202,9 \pm 3,8$	$1187,6 \pm 2,8$	$1182,1 \pm 2,8$	$4780,6 \pm 11,4$

Tabela 7: Duração em minutos das consultas utilizando o formato Parquet. Os valores em negrito representam execuções onde houve operações de *Sort Merge Join*.

Entrada	Q1.1	Q2.1	Q3.1	Q4.1	Total
Star ($SF = 1$)	$1,3 \pm 1,4$	$1,8 \pm 1,1$	$1,3 \pm 0,3$	$1,9 \pm 0,6$	$6,4 \pm 3,6$
Flat ($SF = 1$)	$1,0 \pm 1,6$	$1,1 \pm 0,8$	$0,8 \pm 0,3$	$0,9 \pm 0,2$	$3,9 \pm 2,9$
Star ($SF = 10$)	$3,7 \pm 2,7$	$5,2 \pm 1,5$	$4,3 \pm 0,3$	$6,4 \pm 0,5$	$19,7 \pm 5,1$
Flat ($SF = 10$)	$3,2 \pm 1,8$	$3,7 \pm 1,0$	$3,1 \pm 0,3$	$3,5 \pm 0,1$	$13,6 \pm 3,1$
Star ($SF = 100$)	$22,9 \pm 2,0$	$40,2 \pm 0,8$	$\mathbf{76,6 \pm 5,9}$	$\mathbf{102,0 \pm 4,8}$	$241,8 \pm 6,5$
Flat ($SF = 100$)	$22,9 \pm 2,1$	$28,3 \pm 0,7$	$27,0 \pm 0,3$	$31,7 \pm 0,2$	$110,0 \pm 3,0$

Em relação à duração, a desnormalização se mostrou uma boa alternativa apenas para o formato Parquet. Pode ser observado na Tabela 6 que mesmo no melhor caso (consulta Q1.1 e fator de escala 1), o tempo de execução do formato CSV desnormalizado foi 3,5 vezes maior que o normalizado. Já em relação ao formato Parquet, o tempo de todas as consultas no esquema desnormalizado foi inferior ao normalizado, chegando a reduzi-lo em mais de 3 vezes, como pode ser visto na Tabela 7. Além disso, a diferença entre o esquema normalizado e desnormalizado no Parquet foi maior nas consultas que tinham operações de SMJ.

Tabela 8: Uso máximo de memória utilizada por todos os *workers* durante a execução de cada consulta. Os valores em negrito representam execuções onde houve operações de *Sort Merge Join*.

	Q1		Q2		Q3		Q4	
Entrada	**CSV**	**Parquet**	**CSV**	**Parquet**	**CSV**	**Parquet**	**CSV**	**Parquet**
Star ($SF = 1$)	14,5G	7,4G	14,7G	7,4G	14,7G	7,4G	14,7G	7,5G
Flat ($SF = 1$)	12,3G	1,3G	12,3G	6,7G	12,4G	7,1G	12,4G	7,1G
Star ($SF = 10$)	37,6G	12,1G	**38,7G**	12,3G	38,8G	12,7G	39,0G	12,7G
Flat ($SF = 10$)	14,8G	11,2G	14,7G	11,2G	14,2G	11,2G	14,8G	11,3G
Star ($SF = 100$)	53,7G	41,4G	**55,4G**	41,4G	**55,4G**	**41,9G**	**56,1G**	**42,2G**
Flat ($SF = 100$)	31,1G	32,5	33,1G	35,2	35,4G	36,1G	36,7G	37,2G

Em relação ao uso de memória, houve uma diminuição em todas as execuções no esquema desnormalizado, como pode ser visto na Tabela 8. Além disso, não houve impacto significativo nas execuções com SMJ.

Tabela 9: Quantidade de dados trafegados nas operações de *shuffle* em cada entrada. Os valores em negrito representam execuções onde houve operações de *Sort Merge Join*.

	Q1.1		Q2.1		Q3.1		Q4.1	
Entrada	**CSV**	**Parquet**	**CSV**	**Parquet**	**CSV**	**Parquet**	**CSV**	**Parquet**
Star ($SF = 1$)	1.3K	1.3K	376.1K	31.7K	262.2K	21.9K	63.8K	5.3K
Flat ($SF = 1$)	1.3K	1.3K	376.1K	63.4K	262.3K	43.7K	63.8K	10.6K
Star ($SF = 10$)	2.6K	1.3K	**1.0G**	206.3K	502.4K	142.6K	122.3K	35.2K
Flat ($SF = 10$)	12.1K	1.6K	3.2M	460.0K	2.2M	316.8K	558.4K	77.5K
Star ($SF = 100$)	26.9K	7.4K	**10.3G**	2.0M	**9.7G**	**9.7G**	**13.8G**	**13.8G**
Flat ($SF = 100$)	120.8K	24.6K	32.1M	6.6M	22.3M	4.5M	5.4M	1.1M

Em relação ao *shuffle* (Tabela 9), a desnormalização obteve resultados mais significativos apenas para execuções onde havia a existência de SMJ. Nesses casos, o esquema desnormalizado teve uma redução de pelo menos 300 vezes em relação ao esquema normalizado.

De forma geral, a desnormalização do Parquet foi melhor que a desnormalização do CSV nas condições avaliadas. Utilizar um arquivo CSV desnormalizado não é uma boa escolha, visto a diminuição da performance apresentada nesse experimento. A denormalização para um arquivo Parquet, no entanto, pode ser uma boa alternativa quando o mesmo arquivo é lido várias vezes e existe mais de uma operação de BHJ ou pelo menos uma de SMJ.

4

CONSIDERAÇÕES FINAIS

Esse trabalho propôs a desnormalização de arquivos utilizados como entrada no Spark SQL. Durante esse processo, foi avaliada a performance de dois formatos de arquivos diferentes, o CSV e o Parquet. Além disso, também foi analizado o tipo de operação de junção utilizada pelo Spark em cada execução, afim de descobrir o impacto desta na performance.

No ambiente proposto, o formato CSV e Parquet tiveram resultados diferentes. A desnormalização do CSV não se mostrou uma boa escolha, pois aumentou a duração da execução em mais de 3,5 vezes. Já a desnormalização do Parquet não impactou na performance em alguns casos, porém em outros, conseguiu executar até 3,2 vezes mais rápido.

Os dados descobertos nesse trabalho devem auxiliar na decisão de modelagem do esquema dos arquivos utilizados no Spark. Além disso, existe a possibilidade também destas descobertas serem utilizadas em outras ferramentas de DW com suporte aos mesmos formatos de arquivo utilizados nesse experimento, como o MapReduce e Hive.

Um problema que surgiu durante a pesquisa foi o tempo de execução para grandes arquivos, uma vez que o experimento consistiu em várias releituras nesses arquivos. Isso impossibilitou o teste de mais variações de consultas durante o desenvolvimento do trabalho.

REFERÊNCIAS

BELOV CIFERRI, R. R. Um benchmark voltado a analise de desempenho de sistemas de informações geograficas. , [S.l.], 1995.

BELOV, V.; TATARINTSEV, A.; NIKULCHEV, E. Choosing a Data Storage Format in the Apache Hadoop System Based on Experimental Evaluation Using Apache Spark. **Symmetry**, [S.l.], v.13, n.2, 2021.

BRAAMS, B. Predicate Pushdown in Parquet and Databricks Spark. In: **Anais...** [S.l.: s.n.], 2018.

CHAMBERS, B.; ZAHARIA, M. **Spark**: the definitive guide big data processing made simple. 1st.ed. [S.l.]: O'Reilly Media, Inc., 2018.

CHENG, Y. et al. OPS: optimized shuffle management system for apache spark. In: INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING - ICPP, 49., New York, NY, USA. **Anais...** Association for Computing Machinery, 2020. (ICPP '20).

DAVIDSON, A. Optimizing Shuffle Performance in Spark. In: ASSOCIATION FOR COMPUTING MACHINERY. **Anais...** [S.l.: s.n.], 2013.

O'NEIL, P.; O'NEIL, B.; CHEN, X. The Star Schema Benchmark (SSB). , [S.l.], 01 2009.

PERRIN, J.-G. **Spark in Action**. 2.ed. [S.l.]: Manning Publications, 2020.

POESS, M.; FLOYD, C. New TPC Benchmarks for Decision Support and Web Commerce. **SIGMOD Rec.**, New York, NY, USA, v.29, n.4, p.64–71, Dec. 2000.

SILVA, Y.; ALMEIDA, I.; QUEIROZ, M. SQL: from traditional databases to big data. In: **Anais...** [S.l.: s.n.], 2016. p.413–418.

ZHANG, W. et al. Shuffle-efficient distributed Locality Sensitive Hashing on spark. In: IEEE CONFERENCE ON COMPUTER COMMUNICATIONS WORKSHOPS (INFOCOM WKSHPS), 2016. **Anais...** [S.l.: s.n.], 2016. p.766–767.