# FUNDAMENTOS DE BIG DATA

SOLUÇÕES EDUCACIONAIS INTEGRADAS

# Hive: consultas por meio da linguagem HiveQL

*Roger Robson dos Santos*

**OBJETIVOS DE APRENDIZAGEM**

- Explicar o processo de consultas por meio de HQL no Apache Hive.
- Ilustrar de maneira prática, por meio de *scripts*, os elementos necessários para a criação de consultas no Apache Hive.
- Diferenciar as linguagens de consulta HQL e SQL.

## Introdução

A análise de grandes volumes de dados sempre foi um desafio para as empresas. Para resolver esse problema, nasceu o Hadoop, uma plataforma que possibilita o armazenamento de grandes volumes de dados com um custo muito menor. Contudo, o Hadoop exigia que seus usuários possuíssem muitos conhecimentos em programação e estivessem familiarizados com a Application Programming Interface (API) do MapReduce. Então, para facilitar os processos, surgiu o Hive, que possibilita a análise de dados no Hadoop por meio de uma linguagem de consulta estruturada com a sintaxe parecida com a da Standard Query Language (SQL), permitindo consultas em grandes volumes de dados.

Neste capítulo, você vai estudar o processo de consulta da HiveQL (HQL) no Apache Hive, vendo exemplos práticos. Além disso, você vai conhecer as principais diferenças entre a HQL e a SQL.

## Processo de consultas por meio de HQL no Apache Hive

Construído em Java, o Hadoop é uma plataforma distribuída desenvolvida para ser utilizada com *clusters* e grandes volumes de dados, com tolerância a falhas voltada para *hardware* comum. A biblioteca Apache Hadoop permite a utilização de modelos simples de programação com processamento distribuído (THE APACHE SOFTWARE FOUNDATION, c2006-2020).

O Apache Hive é uma infraestrutura de análise de dados baseada no Apache Hadoop. O Hive permite facilidades aos usuários nas consultas e análises de grandes volumes de dados. Além disso, oferece locais a fim de integrar funcionalidades para o usuário realizar análises personalizadas. O Hive não tem um formato único para armazenamento: além de ser integrado para arquivos separados por vírgula e tabulação (CSV/TSV), Apache ORC, Apache Parquet, entre outros, também permite que usuários criem formatos (THE APACHE SOFTWARE FOUNDATION, c2011-2014).

Inicialmente, o Hive foi desenvolvido pelo Facebook. Devido ao seu sucesso, ele logo foi adotado por outras empresas, como Amazon, Hortonworks e Netflix. Entre suas funcionalidades, está a conversão das tarefas realizadas pelo MapReduce, executadas em *cluster* Hadoop. O Hive também utiliza estruturas MapReduce para agendamento, análise e paralelização de consultas HQL (CAMACHO-RODRÍGUEZ *et al*., 2019). Localizado na parte superior da estrutura do Hadoop, o Hive fornece consultas e análises de grandes volumes de dados. Além disso, o Hive é o primeiro mecanismo *open source* baseado em SQL criado para o Apache Hadoop (HIRVE; REDDY C.H., 2019).

O Apache Hive facilita a leitura, a gravação e o gerenciamento de grandes volumes de dados que estão disponíveis no armazenamento distribuído; ele pode ser consultado por meio da HQL com a sintaxe da SQL. No Hive, são fornecidas funcionalidades da SQL padrão, uma mistura que inclui SQL-92, MySQL e Oracle. Além disso, ele permite a utilização de códigos e funções definidas pelo usuário. O Hive é uma aplicação projetada para tarefas tradicionais de armazenamento de dados, não sendo eficaz em procedimentos como transações *on-line* (WHITE, 2015).

Para entender melhor a arquitetura do Apache Hive, observe a Figura 1, que ilustra os componentes e suas relações. A seguir, veja a descrição das interfaces ilustradas.

- **Thrift:** permite que o Hive interaja com qualquer linguagem de programação que suporte esse protocolo. Atualmente, existem clientes de terceiro para Python e Ruby.
- **Java DataBase Connectivity (JDBC):** utiliza arquitetura e componentes do Hive, fornecendo um *driver* do tipo 4, ou seja, Java puro para conexão com o servidor do Hive.
- **Open DataBase Connectivity (ODBC):** esse *driver* permite que aplicativos que suportam o protocolo ODBC conectem-se ao Apache Hive. O Hive, por padrão, não vem com um *driver* ODBC, porém é possível adquirir esse *driver* com vários fornecedores gratuitamente.

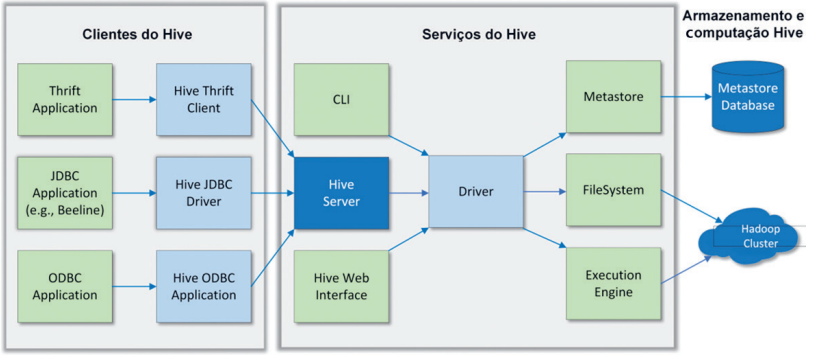


**Figura 1.** Arquitetura do Apache Hive.
**Fonte:** Adaptada de White (2015).

O Hive contém dois componentes. O primeiro é o HCatalog, uma camada que gerencia as tabelas e o armazenamento do Hadoop, permitindo que os usuários utilizem diferentes ferramentas de processamento de dados, como Pig e MapReduce, para ler e gravar dados. O segundo componente é o WebCat, um serviço que executa tarefas no Hadoop MapReduce (ou Yarn), no Pig e no Hive. Ele também executa operações do Hive atrás da interface HTTP (THE APACHE SOFTWARE FOUNDATION, c2011-2014).

A HQL é uma linguagem para consultas semelhante à SQL. Ela é utilizada para consultar e gerenciar grandes volumes de dados. A HQL fornece operações SQL básicas. Tais operações são utilizadas em tabelas ou partições. Essas operações são:

- fazer inserções;
- selecionar colunas de uma tabela usando `SELECT`;
- filtrar linhas de uma tabela utilizando `WHERE`;
- gerenciar tabelas e partições (criar, alterar e eliminar);
- armazenar resultados de uma consulta em um diretório Hadoop ou em outra tabela;
- avaliar a junção de várias colunas agrupadas em uma tabela;
- fazer junções de tabelas;
- baixar o conteúdo de uma tabela para um diretório local.

## Consultas no Apache Hive na prática

Agora você vai conhecer comandos e *scripts* de utilização da HQL. Entre os comandos, estão os de criação, visualização, alteração e eliminação de tabelas e partições utilizando a HQL. Você também vai conhecer os comandos de consulta, *joins*, agregações, inserção de arquivos, inserção de partições dinâmicas, inserção de informações em arquivos locais, amostragem, união de tabelas para realizar consultas, operações com matrizes e *co-groups*.

Imagine que você é funcionário de uma grande empresa e que precisa criar uma tabela chamada `usuario _ ativo` que possua os campos `tempo _ visualizacao`, `id _ usuario`, `ultima _ url _ acessada` e `ip`. Além disso, a tabela deve estar particionada com uma coluna chamada `data`. O objetivo dessa tabela é armazenar dados das atividades de um usuário enquanto ele estiver ativo em algum local da aplicação. Você sabe como criar essa tabela? Veja a seguir.

No exemplo a seguir, é criada a tabela `usuario _ ativo` por meio do comando `CREATE TABLE`. Logo em seguida, são definidos os campos da tabela: `tempo _ visualizacao INT`, `id _ usuario BIGINT`, `ultima _ url _ acessada STRING` e `ip STRING`. Também foi definido o campo `data STRING` com a opção `PARTIONED BY` na coluna `data`. Isso serve para que os dados dessa coluna sejam armazenados separadamente dos demais dados da tabela. Além disso, o exemplo mostra a utilização do comando `COMMENT`, utilizado para adicionar comentários ao código. Por fim, o comando `STORED AS SEQUENCEFILE` sinaliza que os dados serão armazenados em formato binário utilizando Hadoop `SEQUENCEFILE` no Hadoop Distributed File System (HDFS).

**Exemplo**

```
CREATE TABLE usuario _ ativo(tempo _ visualizacao
INT, id _ usuario BIGINT,
                ultima _ url _ acessada STRING, ip STRING
COMMENT 'Endereço de IP do usuario)
  COMMENT 'Esta tabela guarda informações das visualiza-
ções da página'
  PARTITIONED BY (data STRING)
  STORED AS SEQUENCEFILE;
```

A utilização do particionamento permite que as consultas sejam otimizadas por meio de filtros aplicados a colunas específicas. Entretanto, muitos particionamentos podem deixar o DataNode do HDFS mais lento, devido à criação de muitos arquivos e diretórios. Isso faz com que consultas utilizando `GROUP BY` fiquem lentas.

O exemplo a seguir mostra a criação de uma tabela utilizando `ROW FORMAT DELIMITED`, que permite: delimitar os campos com a cláusula 1 utilizando o comando `FIELDS TERMINATED BY '1'`; delimitar a coleção de itens com a cláusula 2 utilizando o comando `COLLECTION ITEMS TERMINATED BY '2'`; e, por fim, delimitar o mapa de chaves com a cláusula 3 utilizando o comando `MAP KEYS TERMINATED BY '3'`.

**Exemplo**

```
CREATE TABLE usuario _ ativo(tempo _ visualizacao
INT, id _ usuario BIGINT,
                ultima _ url _ acessada STRING, ip STRING
COMMENT 'Endereço de IP do usuario)
  COMMENT 'Esta tabela guarda informações das visualiza-
ções da página'
  PARTITIONED BY (data STRING)
  ROW FORMAT DELIMITED
          FIELDS TERMINATED BY '1'
          COLLECTION ITEMS TERMINATED BY '2'
          MAP KEYS TERMINATED BY '3'
  STORED AS SEQUENCEFILE;
```

Outra funcionalidade consiste em agrupar os dados da tabela por meio da coluna `id _ usuario`. Logo em seguida, os dados são colocados em ordem crescente com base na coluna `tempo _ visualizacao`. Por fim, a tabela é agrupada em 10 blocos (*buckets*).

### *Exemplo*

```
CREATE TABLE usuario _ ativo(tempo _ visualizacao
INT, id _ usuario BIGINT,
                ultima _ url _ acessada STRING, ip STRING
COMMENT 'Endereço de IP do usuario)
  COMMENT 'Esta tabela guarda informações das visualiza-
ções da página'
  PARTITIONED BY (data STRING)
  CLUSTERED BY (id _ usuario) SORTED BY (tempo _ visualizacao)
INTO 10 BUCKETS
  ROW FORMAT DELIMITED
          FIELDS TERMINATED BY '1'
          COLLECTION ITEMS TERMINATED BY '2'
          MAP KEYS TERMINATED BY '3'
  STORED AS SEQUENCEFILE;
```

## Comandos para visualizar tabelas

Agora que as tabelas já foram criadas, é necessário utilizar comandos de visualização. O comando a seguir mostra as tabelas existentes:

```
SHOW TABLES;
```

Para mostrar tabelas com um prefixo específico (por exemplo, usuário), como está em jogo uma expressão Java, é necessário colocar o ponto após a palavra e, logo em seguida, o asterisco.

```
SHOW TABLES usuario.*';
```

Use o comando a seguir para mostrar as partições contidas na tabela. Entretanto, se não houver partição, retornará um erro.

```
SHOW PARTITIONS usuario_ativo;
```

Para mostrar as colunas e os tipos de colunas na tabela, utilize:

```
DESCRIBE usuario_ativo;
```

Para mostrar todas as colunas e as propriedades da tabela, utilize o comando a seguir. Normalmente, ele é usado para depuração.

```
DESCRIBE EXTENDED usuario_ativo;
```

Para mostrar todas as colunas e todas as propriedades de uma partição específica, utilize o comando a seguir. Normalmente, ele é usado para depuração.

```
DESCRIBE EXTENDED usuario_ativo PARTITION (data =
'2020-09-01');
```

## Comandos para alterar tabelas

Outro passo importante é realizar alterações nas tabelas. Para renomear uma tabela já existente, utilize o comando a seguir. Caso a tabela não exista, retornará um erro.

```
ALTER TABLE nome_tabela_antiga RENAME TO
nome_tabela_nova;
```

Para renomear colunas em uma tabela já existente, utilize o comando a seguir. É importante usar os mesmos tipos de colunas para evitar erros durante o processo.

```
ALTER TABLE nome_tabela_antiga REPLACE COLUMNS
(coluna1 TIPO, ...);
```

Para adicionar colunas em uma tabela já existente, utilize:

```
ALTER TABLE nome_tabela ADD COLUMNS (coluna1 INT
COMMENT 'Nova coluna do tipo INT');
```

## Comandos para eliminar tabelas e partições

Para eliminar tabelas que não sejam mais importantes para o sistema, use os comandos a seguir.

Para eliminar uma tabela existente, utilize:

```
DROP TABLE nome_tabela;
```

Para eliminar uma partição, altere a tabela e selecione a partição a ser eliminada:

```
ALTER TABLE nome_tabela DROP PARTITION (ds='2020-08-31')
```

O Hive também permite que você crie tabelas externas, que podem apontar para um local específico e ser manipuladas pelos comandos do HDFS.

```
CREATE TABLE usuario_ativo_tabela_antiga(usuario_
visualizacao INT, id_usuario BIGINT,
              ultima_url_acessada STRING, ip STRING
COMMENT 'Endereço de IP do usuario)
ROW FORMAT DELIMITED FIELDS TERMINATED BY '1' LINES
TERMINATED BY '2'
STORED AS TEXTFILE
LOCATION '/home/usuario/bd/usuario_visualizacao';
```

Você também pode adicionar um arquivo à tabela. Para isso, utilize o comando `Hadoop dfs -put`.

```
Hadoop dfs -put arquivo.txt /home/usuario/bd/
usuario_visualizacao
```

Além disso, você pode adicionar dados de uma tabela antiga a uma nova tabela.

```
FROM usuario _ ativo _ tabela _ antiga pvs
INSERT OVERWRITE TABLE usuario _ tabela _ nova
PARTITION(data=2020-08-31')
SELECT pvs.tempo _ visualizacao, pvs.id _ usuario, pvs.
ultima _ url _ acessada, pvs.ip
WHERE pvs.data = '2020-08-31';
```

Ainda é possível inserir os dados carregando-os diretamente de um arquivo que se encontra no servidor.

```
LOAD DATA LOCAL INPATH /home/usuario/
arquivo _ 2020-08-31.txt INTO TABLE usuario _ tabela _ nova
PARTITION(date='2020-08-31')
```

Ademais, é possível apontar um diretório com todos os arquivos que serão carregados; entretanto, não pode haver subdiretórios.

```
LOAD DATA LOCAL INPATH /home/usuario/
arquivo _ 2020-08-31.txt INTO TABLE cliente _ tabela _ nova
PARTITION(date='2020-08-31')
```

Caso você deseje inserir dados que já se encontram no HDFS, a sintaxe ficará como no exemplo a seguir.

```
LOAD DATA INPATH '/usuario/arquivo _ 2020-08-31.txt' INTO
TABLE usuario _ tabela _ nova PARTITION(date='2020-08-31')
```

## Consultas (*query*) ao banco de dados

Agora que você já conhece os comandos de criação de tabelas, vai ver quais são os comandos que pode utilizar nas tabelas já criadas. O primeiro passo é entender as consultas e verificar quais são as suas diferenças em relação à SQL.

Diferentemente do que ocorre na utilização de uma SQL comum, as consultas da HQL fazem sempre uma nova inserção na tabela. Entretanto, como você pode ver no exemplo a seguir, a consulta fica gravada em um arquivo temporário.

```
INSERT OVERWRITE TABLE usuario _ ativo
SELECT usuario.*
FROM usuario;
WHERE usuario.ativo = 1
```

Com o comando `Where`, é possível criar condições nas consultas. Você pode consultar, por exemplo, somente dados do dia 31 de agosto de 2020. Utilizando uma segunda sintaxe, você pode selecionar ainda dados que terminem com o domínio "siteteste.com". Dessa maneira, a consulta determina automaticamente quais partições o sistema vai utilizar com base nas condições impostas no comando `Where`.

```
INSERT OVERWRITE TABLE siteteste _ usuario _ ativo
SELECT usuario _ ativo.*
FROM usuario _ ativo
WHERE usuario _ ativo.data >= '2020-08-01' AND usuario _
ativo.data <= '2020-08-31' AND
     usuario _ ativo.paginas _ acessadas like '%siteteste.
com';
```

## União de consultas (*joins*)

Em diversas atividades de consultas, é necessário unir uma consulta a duas ou mais tabelas. A seguir, veja como realizar a união da tabela `usuario` à tabela `usuario _ ativo`. Esse processo captura o nome e a idade do usuário por meio da coluna `id _ usuario` contida nas duas tabelas. Por fim, a sintaxe `Where` que seleciona apenas dados do dia 31 de agosto de 2020.

```
INSERT OVERWRITE TABLE pv _ usuario
SELECT pv.*, u.nome, u.idade
FROM usuario u JOIN usuario _ ativo pv ON (pv.id _ usuario
= u.id _ usuario)
WHERE pv.data = '2020-08-31';
```

Para realizar consultas completas, utilize o comando `FULL OUTER JOIN`.

```
INSERT OVERWRITE TABLE pv _ usuario
SELECT pv.*, u.nome, u.idade
FROM usuario u FULL OUTER JOIN usuario _ ativo pv ON
(pv.id _ usuario = u.id _ usuario)
WHERE pv.data = '2020-08-31'
```

Para realizar consultas simples que verificam a existência de uma chave em outra tabela, utilize o comando `LEFT SEMI JOIN`.

```
INSERT OVERWRITE TABLE pv _ usuario
SELECT pv.*, u.nome, u.idade
FROM usuario u LEFT SEMI JOIN usuario _ ativo pv ON (pv.
id _ usuario = u.id _ usuario)
WHERE pv.data = '2020-08-31'
```

Para unir várias tabelas em uma consulta, utilize a sintaxe `JOIN`.

```
INSERT OVERWRITE TABLE pv _ usuario
SELECT pv.*, u.nome, u.idade, u.modulo
FROM usuario u JOIN usuario _ ativo u ON (pv.id _ usuario
= u. id _ usuario) JOIN usuario.curso f ON (u.id _ usuario
= f.usuario)
WHERE pv.data = '2020-08-31'
```

## Agregações (*aggregations*)

Agregações são uma excelente otimização na HQL, permitindo consultas mais rápidas e eficazes. No exemplo a seguir, a agregação de dados que permitem a contagem de usuários por cidade se dá por meio do comando `count` junto à consulta.

```
INSERT OVERWRITE TABLE pv _ cidade _ sum
SELECT pv _ usuario.cidade, count (DISTINCT pv _ usuario.
id _ usuario)
FROM pv _ usuario
GROUP BY pv _ usuario.cidade;
```

É permitido, por meio da HQL, utilizar múltiplas agregações. Entretanto, essa consulta não pode ter duas colunas diferentes ao utilizar a sintaxe `DISTINCT`.

```
INSERT OVERWRITE TABLE pv _ cidade _ sum
SELECT pv _ usuario.cidade, count (DISTINCT pv _ usuario.
id _ usuario), count(*), sum(DISTINCT pv _ usuario.
id _ usuario)
FROM pv _ usuario
GROUP BY pv _ usuario.cidade;
```

## Multitabelas/inserções de arquivos (*multi-table/file inserts*)

Realizar a inserção de diversos dados contidos em outras tabelas ou arquivos é uma importante tarefa. A seguir, veja como inserir os dados agrupados de uma tabela antiga em uma nova tabela.

```
FROM pv _ usuario
INSERT OVERWRITE TABLE pv _ cidade _ sum
SELECT pv _ usuario.cidade, count _ distinct(pv _ cidade.
id _ usuario)
GROUP BY pv _ id _ usuario.cidade
```

No exemplo a seguir, os dados de um grupo são inseridos em um arquivo do HDFS.

```
FROM pv _ usuario
INSERT OVERWRITE DIRECTORY '/usuario/data/tmp/
pv _ cidade _ sum'
SELECT pv _ usuario.cidade, count _ distinct(pv _ cidade.
id _ usuario)
GROUP BY pv _ id _ usuario.cidade
```

## Inserção em partições dinâmicas (*dynamic-partition insert*)

A HQL permite realizar inserções em diversas partições de forma dinâmica. Considere o exemplo a seguir, em que foram adicionadas várias partições da tabela `usuario_ativo`.

**Exemplo**

```
FROM usuario _ ativo _ stg pvs
   INSERT OVERWRITE TABLE usuario _ ativo
PARTITION(data='2020-08-10')
  SELECT pvs.tempo _ visualizacao, pvs.id _ usuario, pvs.
ultima _ url _ acessada, pvs.ip WHERE pvs.data = '2020-08-10'
  INSERT OVERWRITE TABLE usuario _ ativo
PARTITION(data='2020-08-20')
  SELECT pvs.tempo _ visualizacao, pvs.id _ usuario, pvs.
ultima _ url _ acessada, pvs.ip WHERE pvs.data = '2020-08-20'
  INSERT OVERWRITE TABLE usuario _ ativo
PARTITION(data='2020-08-30')
  SELECT pvs.tempo _ visualizacao, pvs.id _ usuario, pvs.
ultima _ url _ acessada, pvs.ip WHERE pvs.data = '2020-08-30';
```

No exemplo anterior, mesmo que uma partição não exista, ela será automaticamente criada. No exemplo a seguir, veja como seria adicionar todas as partições sem definir a data de cada uma delas.

**Exemplo**

```
FROM usuario _ ativo _ stg pvs
   INSERT OVERWRITE TABLE usuario _ ativo
PARTITION(data)
  SELECT pvs.tempo _ visualizacao, pvs.id _ usuario, pvs.
ultima _ url _ acessada, pvs.ip WHERE pvs.data = '2020-08-10'
```

## Inserção de informações em arquivos locais

Para inserir informações de saída em arquivos locais no servidor, utilize:

```
INSERT OVERWRITE LOCAL DIRECTORY '/tmp/pv _ cidade _ sum'
SELECT pv _ cidade _ sum.*
FROM pv _ cidade _ sum;
```

## Amostragem

O Hive permite que o usuário escolha amostragens dentro da tabela. Para isso, o usuário escolhe o bloco do qual quer coletar os dados a fim de adicioná-los a uma nova tabela. Anteriormente, você viu a criação de tabelas utilizando a

sintaxe `CLUSTERED BY`, que define quantos blocos terá a tabela. A sintaxe de consulta, como você pode ver a seguir, mostra que *X* é o bloco que o usuário quer coletar, enquanto *Y* é o total de blocos contidos na tabela. Entretanto, o *X* deve ser o equivalente múltiplo do valor *Y*.

```
TABLESAMPLE(BUCKET X OUT OF Y)
```

No exemplo a seguir, veja a coleta do terceiro bloco em uma tabela com 16 blocos.

***Exemplo***

```
INSERT OVERWRITE TABLE pv _ cidade _ sum _ sample
   SELECT pv _ cidade _ sum.*
FROM pv _ cidade _ sum TABLESAMPLE(BUCKET 3 OUT OF 16);
```

## União de todas as tabelas em uma consulta (*union all*)

É possível unir diversas tabelas para gerar uma única consulta. No exemplo a seguir, é realizada uma consulta à tabela `historico _ postagem`, para coletar as postagens do usuário. Logo em seguida, é consultada a tabela `historico _ comentario`, para capturar os comentários do usuário. Por fim, é feita a união dos conteúdos à tabela `usuario`.

***Exemplo***

```
INSERT OVERWRITE TABLE usuario _ historico
   SELECT u.id, historico.data
FROM (
     SELECT av.uid AS uid
     FROM historico _ postagem av
     WHERE av.data = '2020-08-31'

     UNION ALL

     SELECT ac.uid AS uid
     FROM historico _ comentario ac
     WHERE ac.data = '2020-08-31'
     ) historico JOIN usuario u ON(u.id = historico.uid);
```

## Operações com matriz (*array operations*)

Caso o usuário tenha cadastrado uma matriz no banco de dados, ele pode coletar elementos da matriz especificando a coluna e a posição. No exemplo a seguir, veja a coleta do elemento 5 da matriz, sabendo que uma matriz se inicia na posição 0.

```
SELECT pv.comentarios[4]
FROM historico _ comentario pv;
```

Também é possível coletar o tamanho de uma matriz. Veja no exemplo a seguir.

```
SELECT pv.id _ usuario, size(pv.comentarios)
FROM historico _ comentario pv;
```

## Operações de mapa — matrizes associativas (*map — associative arrays — operations*)

Essa matriz tem o propósito de selecionar uma propriedade do Map, como `[String, String]`. Veja o exemplo a seguir, que mostra a construção de uma consulta selecionando uma propriedade `tipo _ comentario` de uma matriz.

```
INSERT OVERWRITE usario _ ativo _ temp
SELECT    pv.id _ usuario,    pv.propriedades
['tipo _ comentario']
FROM usario _ ativo pv;
```

Além disso, é possível coletar o tamanho da matriz, como você pode ver no exemplo a seguir.

```
SELECT site(pv.propriedades)
FROM usario _ ativo pv;
```

### *Co-groups*

Uma operação importante no Hive é o *co-group*, que possibilita o envio de dados de várias tabelas para um redutor personalizado com várias linhas agrupadas por valores de certas colunas nas tabelas. Para essa união, utilize a sintaxe `UNION ALL` com `CLUSTER BY`. Suponha que você tem as tabelas `historico _ postagem` e `historico _ comentario` e deseja agrupá-las na coluna `uid`, enviando-as logo em seguida a um *script* personalizado (script01_personalizado), com a sintaxe especificada. Veja esse exemplo a seguir.

**Exemplo**

```
FROM (
                FROM (
             FROM historico _ postagem av
           SELECT av.uid AS uid, av.id AS id, av.data AS data

             UNION ALL

             FROM historico _ comentario ac
             SELECT ac.uid AS uid, ac.id AS id, ac.data AS
data
        ) union _ historicos
        SELECT union _ historicos.uid, union _ actions.id,
union _ actions.data
        CLUSTER BY union _ historicos.uid) map

        INSERT OVERWRITE TABLE historicos _ personalizados
        SELECT TRANSFORM(map.uid, map.id, map.data) USING
'script01 _ personalizado' AS (uid, id, reduced _ val);
```

## Principais diferenças entre a HQL e a SQL

Apesar de a HQL e a SQL serem muito semelhantes, essas duas linguagens de consulta têm inúmeras diferenças entre si. A SQL é utilizada em gerenciamento de bancos de dados relacionais, também conhecidos como *Relational Database Management Systems* (RDBMS). Ou seja, a SQL é uma linguagem utilizada para armazenar, manipular e recuperar dados em bancos de dados. Já a HQL é uma linguagem de consulta utilizada no Hive que permite analisar e processar dados semiestruturados. Essa linguagem é muito semelhante à

SQL e é altamente escalonável. Ela trabalha com bancos de dados estruturados e suporta quatro tipos de arquivo: *text file*, *sequence file*, ORC e RC *file*.

A SQL trabalha com pequenos conjuntos de dados. Embora essa linguagem proporcione um armazenamento de dados significativo, ela se desenvolve melhor com conjuntos menores de dados. Além disso, a SQL é uma opção perfeita para dados que necessitam ser constantemente atualizados. Quanto à otimização, a SQL pode ser configurada para ter melhor desempenho em matéria de tempo e frequência; ela lida com aplicativos que usam processamento analítico *on-line* (*On-line Analytical Processing* — Olap).

A HQL, por sua vez, é utilizada para consultas analíticas no Apache Hive, pois ela lida facilmente com certas particularidades. Com a HQL, trabalha-se estritamente com estruturas de dados, e o principal objetivo é executar processamento e análises complexas em grandes volumes de dados com facilidade, além de executar consultas para fins analíticos sem dificuldades.

No Quadro 1, veja as principais diferenças entre a HQL e a SQL.

**Quadro 1.** Diferenças entre a HQL e a SQL

| Diferenças | HQL | SQL |
|---|---|---|
| Tipo de dados com que trabalha | Estrutura de dados (*data structure*) | Dados relacionais (*relational data*) |
| Tipo de utilização | Processa grandes volumes de dados | Usada para respostas em tempo real a partir de uma única máquina |
| Armazenamento de dados no *cache* da memória | Não | Sim |
| Sobrecarga de trabalho | Muito alta | Normal |
| Suporte à XML | Não | Sim |
| Principal objetivo | Gerenciar transações e consultas analíticas | Analisar dados |
| MapReduce | Suporta | Não suporta |
| Inserção multitabela | Suporta | Não suporta |

*(Continua)*

*(Continuação)*

| Diferenças | HQL | SQL |
|---|---|---|
| Implementação | Java | C++ |
| Relacionamentos | Considera a relação entre dois objetos | Considera a relação entre duas tabelas |
| Tipos de dados | Booleanos, inteiros, ponto flutuante, ponto fixo, temporais, texto e *strings* binárias, *arrays*, *map*, *struct* | Inteiros, ponto flutuante, ponto fixo, temporais, texto e *strings* binárias |
| *Select* | SQL-92, `SORT BY` para ordenação parcial, `LIMIT` para limitar o número de linhas retornado | SQL-92 |
| *Views* | Suporta atualização a partir do Hive 3 | Suporta atualização |
| *Joins* | *Inner joins*, *map joins*, *cross joins*, *outer joins*, *semi joins* | SQL-92 ou variantes (*join* de tabelas na cláusula `FROM` e condição do *join* na cláusula `WHERE`) |
| *Subqueries* | `FROM`, `WHERE` e `HAVING` | Em qualquer sintaxe |

**Fonte:** (WHITE, 2015), e (BEAULIEU, 2005).

# Referências

BEAULIEU, A. *Learning SQL*. Sebastopol, CA: O'Reilly, 2005.

CAMACHO-RODRÍGUEZ, J. *et al.* Apache Hive: from MapReduce to enterprise-grade Big Data Warehousing. *In:* INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, 19., 2019, Amsterdam. *Anais* [...]. Amsterdam: [SIGMOD], 2019. Disponível em: https://arxiv.org/pdf/1903.10970v1.pdf. Acesso em: 29 set. 2020.

HIRVE, S.; REDDY C.H., Pradeep. A survey on visualization techniques used for Big Data analytics. *In:* BHATIA, S. *et al.* (Ed.). *Advances in computer communication and computational sciences.* Singapore: Springer, 2019. p. 447-459.

THE APACHE SOFTWARE FOUNDATION. *Apache Hadoop.* [Wakefield, Massachusetts]: The Apache Software Foundation, c2006-2020. Disponível em: https://hadoop.apache.org/. Acesso em: 2 out. 2020.

THE APACHE SOFTWARE FOUNDATION. *Apache Hive.* Version 2.0. [Wakefield, Massachusetts]: The Apache Software Foundation, c2011-2014. Disponível em: https://cwiki.apache.org/confluence/display/Hive/Home. Acesso em: 29 set. 2020.

WHITE, T. *Hadoop:* the definitive guide. 4th ed. Sebastopol, CA: O'Reilly, 2015.

## Leituras recomendadas

BENGFORT, B.; KIM, J. *Data Analytics with Hadoop:* an introduction for Data Scientists. Sebastopol: O'Reilly, 2016.

CAPRIOLO, E.; WAMPLER, D.; RUTHERGLEN, J. *Programming Hive:* data warehouse and query language for Hadoop. Sebastopol: O'Reilly, 2012.

SHAW, S. *et al. Practical Hive:* a guide to Hadoop's Data Warehouse System. New York: Apress, 2016.

### *Fique atento*

Os *links* para *sites* da *web* fornecidos neste capítulo foram todos testados, e seu funcionamento foi comprovado no momento da publicação do material. No entanto, a rede é extremamente dinâmica; suas páginas estão constantemente mudando de local e conteúdo. Assim, os editores declaram não ter qualquer responsabilidade sobre qualidade, precisão ou integralidade das informações referidas em tais *links*.

Conteúdo:

SAGAH
SOLUÇÕES EDUCACIONAIS INTEGRADAS