Hive – Escalando um Data Warehouse de um Petabyte usando Hadoop

Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Antony, Hao Liu and Raghotham Murthy

Facebook Data Infrastructure Team

Abstract - O tamanho dos conjuntos de dados coletados e analisados na indústria de business intelligence está crescendo rapidamente, soluções tradicionais de armazenagem são proibitivamente caras.
Hadoop [1] é uma implementação popular open-source do map reduce nas quais está sendo usado em empresas como Yahoo, Facebook etc para armazenar e processar conjuntos de dados extremamente grandes em um hardware. No entanto, o modelo de programação de map reduce é muito baixo nível e requer desenvolvedores para escrever programas personalizados que são difíceis de manter e reutilizar. Neste artigo, apresentaremos o Hive, uma solução open-source de data warehousing construída sobre Hadoop. O Hive suporta consultas expressamente em Linguagem SQL declarativa - HiveQL, que são compilados em jobs map-reduce executados usando o Hadoop. Além disso, HiveQL permite aos usuários conectar scripts personalizados para consultas map-reduce.
A linguagem inclui um tipo de sistema que dá suporte para tabelas contendo tipos primitivos, coleções como arrays e maps, e composições aninhadas. As bibliotecas de IO subjacentes podem ser estendidas para consultar dados em formatos personalizados. Hive também inclui um sistema de catálogo - Metastore - que contém esquemas e estatísticas, que são úteis na exploração de dados, consultas otimizadas e compilação de consultas. No Facebook, o Hive warehouse contém dezenas de milhares de tabelas e armazena 700TB de dados e está sendo usado extensivamente tanto para relatórios e análises ad-hoc por mais de 200 usuários por mês.

I. Introdução

Analisar a escalabilidade em grandes conjuntos de dados tem sido função de várias equipes no Facebook – tanto de engenheiros e não-engenheiros. Além da análise ad hoc aplicativos de business intelligence são usados por analistas na empresa, um número de produtos do Facebook também são baseados nestas análises. Esses produtos variam de relatórios simples como um insight para uma rede de anúncios do Facebook, para um tipo mais avançado tal como o produto da Lexicon no Facebook[2]. Como resultado, uma infra-estrutura flexível que atenda a estas diversas aplicações e utilizações de uma forma rentável com as quantidades cada vez maiores de dados gerados no Facebook, é fundamental. Hive e Hadoop são as tecnologias que temos utilizado para esses requisitos no Facebook.
Toda a infra-estrutura de processamento de dados no Facebook anteriores a 2008 foram construídos em torno de data warehouse construído com RDBMS comerciais. Os dados que estávamos gerando cresceram muito rápido - como um exemplo nós crescemos a partir de 15TB de dados em 2007 para um conjunto de dados de 700TB hoje. A infra-estrutura na época era tão insuficiente que alguns dados diários demorava mais de um dia para ser processados e a situação estava piorando a cada dia que passava. Tínhamos uma necessidade urgente de infra-estrutura que pudesse escalar dados. Como resultado, começamos a explorar o Hadoop como uma tecnologia para atender às nossas necessidades de dimensionamento. O fato de Hadoop ter sido um projeto de código aberto que estava sendo usado em escala de petabyte e proporcionando escalabilidade usando hardware foi uma proposta muito atraente para nós. Os mesmos trabalhos que levavam mais de um dia para ser concluídos poderiam agora ser executados dentro de algumas horas usando Hadoop.
No entanto, o uso do Hadoop não foi fácil para os usuários finais, especialmente para aqueles usuários que não estavam familiarizados com o map-reduce. Os usuários finais tinham que escrever programas para tarefas simples como obter contagens ou médias brutas. Hadoop não tem a

expressividade das linguagens de consulta populares como SQL e, como resultado, os usuários acabavam gastando horas (se não dias) para escrever programas para uma análise simples. Isso foi muito claro para nós, para que realmente pudéssemos capacitar a empresa para analisar esses dados de forma mais produtiva, nós tínhamos que melhorar a capacidade de consulta do Hadoop. Aproximar esses dados dos usuários foi o que nos inspirou a construir o Hive em janeiro de 2007. Nossa visão era trazer os conceitos conhecidos de tabelas, colunas, partições e um subconjunto de SQL para o mundo não-estruturado do Hadoop, mantendo a extensibilidade e flexibilidade que o Hadoop trás. Hive foi disponibilizado em agosto de 2008 e desde então tem sido utilizado e explorado por inúmeros usuários do Hadoop para suas necessidades de processamento de dados.

Desde o início, Hive era muito popular entre todos os usuários no Facebook. Hoje, executamos regularmente milhares de jobs sobre o cluster Hadoop / Hive com centenas de usuários para uma variedade de aplicações iniciando por uma simples sumarização de jobs para inteligência de negócios, aplicações de aprendizagem de máquinas e também para apoiar recursos de produtos do Facebook.

Nas seções a seguir, fornecemos mais detalhes sobre a arquitetura e os recursos do Hive. A Seção II descreve o modelo de dados, os sistemas de tipos e o HiveQL. Seção III detalha como os dados das tabelas do Hive são armazenados no subjacente sistema de arquivos distribuídos - HDFS (sistema de arquivos Hadoop). Seção IV descreve a arquitetura do sistema e vários componentes do Hive. Na Seção V destacamos as estatísticas de uso de Hive no Facebook e fornecemos trabalhos relacionados na Seção VI. Nós concluímos com trabalhos futuros na Seção VII.

II. MODELO DE DADOS, TIPOS DO SISTEMA E LINGUAGEM DE CONSULTA

Hive estrutura os dados no banco de dados em conceitos entendidos como tabelas, colunas, linhas e partições. Suporta todos os principais tipos primitivos - integers, floats, doubles e strings - bem como tipos complexos como maps, lists e structs. Este último pode ser aninhado arbitrariamente para construir tipos complexos. Além disso, o Hive permite aos usuários estender o sistema com seus próprios tipos e funções. A linguagem de consulta é muito semelhante ao SQL e, portanto, pode ser facilmente entendido por qualquer pessoa familiarizada com o SQL. Há alguns modelos de dados, tipo de sistema e HiveQL que são diferentes das bases de dados tradicionais e que foram motivado por nossas experiências adquiridas no Facebook. Nós vamos destacar estes e outros detalhes nesta seção.

A. Modelo de Dados e Tipos do Sistema

Semelhante aos bancos de dados tradicionais, o Hive armazena dados em tabelas, onde cada tabela consiste em um número de linhas, e cada linha consiste em um número especificado de colunas. Cada coluna tem um tipo associado. O tipo é um tipo primitivo ou um tipo complexo. Atualmente, os seguintes tipos primitivos são suportados:

- Inteiros – bigint(8 bytes), int(4 bytes), smallint(2 bytes), tinyint(1 byte). Todos os tipos de inteiro são assinados.
- Números de ponto flutuante - float(single precision), double(double precision), etc.
- String.

Hive também suporta nativamente os seguintes tipos complexos:

- Arrays Associativos – map<key-type, value-type>
- Listas – list<element-type>
- Structs – struct<file-name: field-type, ... >

Estes tipos complexos são modelados e podem gerar tipos de complexidade arbitrária.Por exemplo, list<map<string, struct<p1:int, p2:int>> representa uma lista de arrays associativos que mapeiam strings para estruturas que por sua vez contêm dois campos inteiros chamados p1 e p2. Estes podem estar todos em uma instrução create table para criar tabelas com o esquema

desejado. Por exemplo, a seguinte instrução cria uma tabela t1 com um esquema complexo.

```
CREATE TABLE t1(st string, fl float, li list<map<string, struct<p1:int, p2:int>>);
```

As expressões de consulta podem acessar campos dentro das estruturas usando o operador '.' . Os valores nas matrizes e listas associativas podem ser acessado usando o operador '[]'. No exemplo anterior, t1.li[0] retorna o primeiro elemento da lista e t1.li [0]['chave'] retorna a estrutura associativa com a 'chave' nesse array associativo. Finalmente O campo p2 desta estrutura pode ser acessado por t1.li[0]['chave'].p2. Com estas construções o Hive é capaz de suportar estruturas de complexidade arbitrária.

As tabelas criadas da maneira descrita acima são serializadas e desserializadas usando serializadores e deserializadores já presentes em Hive. Contudo, existem casos em que os dados de uma tabela são preparados por alguns outros programas ou podem mesmo ser dados herdados. O Hive fornece flexibilidade para incorporar esses dados em uma tabela sem transformar os dados, o que pode poupar uma quantidade de tempo para grandes conjuntos de dados. Como descreveremos nas seções anteriores, isso pode ser conseguido através do fornecimento de um jar que implementa a interface SerDe java para o Hive. Em tais situações, a informação de tipo também pode ser fornecida por esse jar proporcionando uma implementação correspondente da interface ObjectInspector e expondo a implementação através do método getObjectInspecto presente na interface SerDe. Mais detalhes sobre essas interfaces pode ser encontrado no wiki do Hive [3], mas a base aqui é que qualquer formato de dados arbitrário e tipos codificados nele podem ser conectados ao Hive, fornecendo um jar que contém implementações para as interfaces SerDe e ObjectInspector.

Todos os SerDes nativos e tipos complexos suportados no Hive são também implementações dessas interfaces. Como resultado das associações apropriadas foram feitos entre a tabela e o jar, a camada de consulta trata estes pares com os tipos nativos e formatos. Como exemplo, a seguinte instrução adiciona um jar contendo as interfaces SerDe e ObjectInspector para o cache distribuído ([4]) para que ele esteja disponível para o Hadoop e então proceda a criar a tabela com o serde personalizado.

```
add jar /jars/myformat.jar;
CREATE TABLE t2
ROW FORMAT SERDE
'com.myformat.MySerDe';
```

Observe que, se possível, o esquema da tabela também pode ser fornecido compondo os tipos complexos e primitivos.

B. Linguagem Query

A linguagem de consulta Hive (HiveQL) é composta por um subconjunto de SQL e algumas extensões que úteis que encontramos em nosso meio ambiente. Os recursos tradicionais do SQL, como da cláusula sub-query, vários tipos de joins - inner, left outer, right outer e outer joins, produtos cartesianos, group bys e aggregations, union all, create table como select e muitas funções úteis em tipos primitivos e complexos tornam a linguagem muito semelhante com o SQL. De fato para muitas das construções mencionadas antes esta é exatamente como o SQL. Isso permite que qualquer pessoa familiarizada com SQL possa iniciar um hive cli (interface de linha de comando) e começar a consultar o sistema imediatamente. Recusros úteis de navegação de metadados exibindo tabelas e descrições também estão presentes e são explicados como recursos do plano para inspecionar os planos que parecem muito diferentes do que você veria em um RDBMS tradicional.Existem algumas limitações, e.g. só predicados de igualdade são suportados em um predicado de junção e o devem ser especificadas usando a sintaxe de junção ANSI como

```
SELECT t1.a1 as c1, t2.b1 as c2
FROM t1 JOIN t2 ON (t1.a2 = t2.b2);
```

Em vez da mais tradicional

```
SELECT t1.a1 as c1, t2.b1 as c2
FROM t1, t2 WHERE t1.a2 = t2.b2;
```

Outra limitação é na forma como as inserções são feitas. Hive atualmente não suporta a inserção em uma tabela existente ou dados particionados e todas as inserções substituem os dados existentes.

Consequentemente, tornamos isso explícito na nossa sintaxe da seguinte forma:

```
INSERT OVERWRITE TABLE t1
SELECT * FROM t2;
```

Na realidade, essas restrições não têm sido um problema. Nós temos raramente um caso em que a consulta não pode ser expressa como equi-join e uma vez que a maioria dos dados é carregada e amazenada diariamente ou de hora em hora, simplesmente carregamos os dados em partição da tabela para esse dia ou hora. No entanto, fazemos perceber que com cargas mais freqüentes o número de partições podem tornar-se muito grandes e que podem exigir que implementemos INSERT INTO semânticos. A falta de INSERT INTO, UPDATE e DELETE no Hive, por outro lado, nos permitem usar mecanismos muito simples para lidar com leitura e escrita concorrentes sem implementar protocolos de bloqueio complexo.

Para além destas restrições, o HiveQL tem extensões para suporte a análise expressa de programas map-reduce por usuários e na linguagem de programação da escolha deles. Isso permite que usuários avançados expressem lógicas complexas em termos de programas map-reduce que são conectados em consultas HiveQL perfeitamente. Algumas vezes isso pode ser uma abordagem única. No caso em que existam bibliotecas em python ou php ou qualquer outra linguagem que o usuário deseja usar para transformação de dados. O exemplo canônico de contagem de palavras em uma tabela de documentos pode, por exemplo, ser expressa usando mapreduce da seguinte maneira:

```
FROM (MAP doctext USING 'python wc_mapper.py' AS (word, cnt) FROM docs CLUSTER BY word) a REDUCE word, cnt USING 'python wc_reduce.py';
```

Conforme mostrado neste exemplo, a cláusula MAP indica que colunas de entrada (doctext neste caso) podem ser transformadas pelo programa (neste caso 'python wc_mapper.py') na saída (palavra e cnt). A cláusula CLUSTER BY especifica uma sub-consulta as colunas de saída que são feitas para distribuir os dados aos redutores e finalmente o REDUCE esta cláusula especifica o programa de usuário a ser invocado (python Wc_reduce.py neste caso) nas colunas de saída da sub-consulta.

Às vezes, os critérios de distribuição entre os mapeadores e os redutores precisam fornecer dados aos redutores de modo ordenado em um conjunto de colunas que são diferentes do que são usados para fazer a distribuição. Um exemplo poderia ser o caso em que todas as ações em uma sessão precisam ser ordenados pelo tempo. Hive fornece as cláusulas DISTRIBUTE BY e SORT BY para fazer isso como mostrado no exemplo:

```
FROM (
 FROM session_table
 SELECT sessionid, tstamp, data
 DISTRIBUTE BY sessionid SORT BY tstamp
) a
REDUCE sessionid, tstamp, data USING
'session_reducer.sh';
```

Observe, no exemplo acima não há nenhuma cláusula de MAP o que indica que as colunas de entrada não são transformadas. Da mesma forma, é possível ter uma cláusula MAP sem um REDUCE caso a fase de redução não faça qualquer transformação de dados. Também nos exemplos mostrados acima, a cláusula FROM aparece antes da cláusula SELECT que é outro desvio da sintaxe SQL padrão. Hive permite usuários trocar a ordem do FROM e SELECT / MAP / REDUCE dentro de uma determinada sub-consulta. Isto torna-se particularmente útil e intuitivo para inserções múltiplas. O HiveQL suporta a inserção de transformações em diferentes tabelas, partições, hdfs ou diretórios locais como parte da mesma consulta. Essa habilidade ajuda na redução do número de varreduras efetuadas nos dados mostrado no exemplo a seguir:

```
FROM t1
 INSERT OVERWRITE TABLE t2

 SELECT t3.c2, count(1)
 FROM t3
 WHERE t3.c1 <= 20
 GROUP BY t3.c2

INSERT OVERWRITE DIRECTORY '/output_dir'
SELECT t3.c2, avg(t3.c1) FROM t3
WHERE t3.c1 > 20 AND t3.c1 <= 30
GROUP BY t3.c2

INSERT OVERWRITE LOCAL DIRECTORY
'/home/dir'
SELECT t3.c2, sum(t3.c1)
FROM t3
WHERE t3.c1 > 30
GROUP BY t3.c2;
```

Neste exemplo, porções diferentes da tabela t1 são agregadas e usadas para gerar uma tabela t2, no diretório hdfs (/output_dir) e no diretório local (/ home/dir da máquina do usuário).

III. ARMAZENAMENTO DE DADOS, SERDE E FORMATOS DE ARQUIVOS

A. ARMAZENAMENTO DE DADOS

Enquanto as tabelas são unidades de dados lógicas em Hive, metadados associativos de tabelas são diretórios do hdfs. As unidades de dados primárias e seus mapeamentos no hdfs são como segue:

• Tables - Uma tabela é armazenada em um diretório em hdfs.

• Partitions - Uma partição da tabela é armazenada em um sub-diretório dentro do diretório de uma tabela.

• Buckets - Um buckets é armazenado em um arquivo dentro do diretório da partição ou tabela, dependendo se a tabela é uma tabela particionada ou não.

Como exemplo, uma tabela test_table é mapeada para `<warehouse_root_directory>/test_table` no hdfs. O `warehouse_root_directory` é especificado pelo parâmetro de configuração `hive.metastore.warehouse.dir` em `Hive-site.xml`. Por padrão, o valor deste parâmetro é definido como /user/hive/warehouse.

Uma tabela pode ser particionada ou não-particionada. A particionada pode ser criada especificando a tabela PARTITIONED BY na instrução CREATE TABLE como mostrado abaixo.

```
CREATE TABLE test_part(c1 string, c2 int)
PARTITIONED BY (ds string, hr int);
```

No exemplo mostrado acima as partições de tabela serão armazenadas no diretório /user/hive/warehouse/test_part no hdfs. Uma partição existe para cada valor distinto de ds e hr específico do usuário. Observe que as colunas de particionamento não fazem parte dos dados da tabela e os valores da coluna de partição são codificados no diretório daquela partição (eles também são armazenados na tabela metadata). Uma nova partição pode ser criada através de um INSERT ou através de uma instrução ALTER que adiciona uma partição para a tabela. As duas afirmações a seguir

```
INSERT OVERWRITE TABLE
test_part PARTITION(ds='2009-01-01', hr=12)
SELECT * FROM t;

ALTER TABLE test_part
ADD PARTITION(ds='2009-02-02', hr=11);
```

Adicionada uma nova partição à tabela test_part. A declaração INSERT também preenche a partição com dados da tabela t,onde a tabela alterada cria uma partição vazia. Essas duas declarações acabam criando os diretórios correspondentes - /user/hive/warehouse/test_part/ds=2009 -01-01/hr=12 e /user/hive/warehouse/ test_part/ds=2009-02-02/hr=11 – na tabela do diretório hdfs. Esta abordagem cria complicações no caso de o valor da partição conter caracteres como / ou: que são usados por hdfs para denotar a estrutura de diretório, mas a fuga adequada cuida para produzir um nome de diretório compatível com hdfs.

O compilador Hive é capaz de usar esta informação para podar os diretórios que precisam ser verificados para obter dados para avaliar uma consulta. No caso da tabela test_part, a consulta

```
SELECT * FROM test_part WHERE ds='2009-
01-01';
```

Digitalizarão todos os arquivos dentro do /user/hive/warehouse/test_part/ds= 2009-01-01 e a consulta

SELECT * FROM test_part WHERE ds='2009-02-02' AND hr=11;

Digitalizarão todos os arquivos dentro do /user/hive/warehouse/test_part/ds = 2009-01-01/hr=12 diretório. A poda dos dados tem um impacto significativo no tempo necessário para processar a consulta. Em muitos esquemas de particionamento é semelhante ao que foi referido como uma lista de particionamento por muitos fornecedores de banco de dados ([6]), mas há diferenças nas quais os valores das chaves de partição são armazenados com os metadados em vez dos dados.

O conceito final de unidade de armazenamento que Hive usa é o conceito Buckets. Um bucket é um arquivo dentro do diretório de nível de folha de uma tabela ou uma partição. No momento em que a tabela é criada, o usuário pode especificar o número de buckets necessários e a coluna buckets para os dados. Na implementação atual, estas informações são usadas para podar os dados caso a consulta do usuário sobre uma amostra de dados, por exemplo, uma tabela bucket com 32 buckets possa gerar rapidamente uma amostra de 1/32, escolhendo olhar para o primeiro bucket de dados. Da mesma forma, que a declaração

SELECT * FROM t TABLESAMPLE(2 OUT OF 32);

Digitalizar os dados presentes no segundo bucket. Observe que o ônus de garantir que os arquivos de bucket sejam devidamente criados e nomeados são uma responsabilidade do aplicativo e as instruções HiveQL DDL atualmente não tentam distribuir os dados em um modo que o torna compatível com as propriedades da tabela. Consequentemente, deve se ter cuidado com as informações bucketeadas.

Embora os dados correspondentes a uma tabela residam no diretório <warehouse_root_directory>/test_table no hdfs, o Hive também permite aos usuários consultar dados armazenados em outros locais do hdfs. Isto pode ser conseguido através da TABLE como mostrado no exemplo a seguir.

CREATE EXTERNAL TABLE test_extern(c1 string, c2 int)
LOCATION '/user/mytables/mydata';

Com essa instrução, o usuário pode especificar que test_extern é uma tabela externa onde cada linha compreende duas colunas - c1 e c2. Além disso, os arquivos de dados são armazenados no local /user/mytables/mydata no hdfs. Observe que, como não foi definido, presume-se que os dados estejam em um formato interno Hive. Uma tabela externa difere de uma tabela normal apenas pelo comando drop table que em uma tabela externa somente descarta os metadados da tabela e não exclui nenhum dado. Um drop em uma tabela normal, por outro lado, exclui os dados associados com a tabela também.

B. Serialização / Deserialização (SerDe)

Como mencionado anteriormente, o Hive dá uma interface java SerDe fornecida ao usuário e associado a uma tabela ou partição. Como resultado, os formatos de dados facilitam a interpretação e a consulta. O padrão SerDe implementado em Hive é chamado de LazySerDe - pois deserializa linhas em objetos internos para que a deserialização de uma coluna só tenha efetividade se a coluna da linha seja necessária em alguma expressão de consulta. O LazySerDe assume que os dados são armazenados no arquivo de forma que as linhas são delimitados por uma nova linha (código ascii 13) e as colunas dentro de uma linha são delimitados por ctrl-A (código ascii 1). Este SerDe também pode ser usado para ler dados que usam qualquer outro delimitador entre as colunas.
Como exemplo, a declaração

CREATE TABLE test_delimited(c1 string, c2 int)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\002'
LINES TERMINATED BY '\012';

Especifica que os dados para a tabela test_delimited usam ctrl-B (código ascii 2) como um delimitador de coluna e usa ctrl-L (código ascii 12) como um delimitador de linha. Além disso, os

delimitadores podem ser especificados para delimitar as chaves serializadas e valores de mapas e diferentes delimitadores também podem ser especificados para delimitar elementos de uma lista (coleção). Isto é ilustrado declaração seguinte.

```
CREATE TABLE test_delimited2(c1 string,
c2 list<map<string, int>>)
 ROW FORMAT DELIMITED
 FIELDS TERMINATED BY '\002'
 COLLECTION ITEMS TERMINATED BY '\003'
 MAP KEYS TERMINATED BY '\004';
```

Além de LazySerDe, alguns outros SerDes interessantes estão presentes no hive_contrib.jar que é fornecido com a distribuição. Um particularmente útil é RegexSerDe que permite ao usuário especificar uma expressão regular para analisar várias colunas fora de uma linha. A seguinte declaração pode ser usado por exemplo, para interpretar logs do apache.

```
add jar 'hive_contrib.jar';
CREATE TABLE apachelog(
host string,
identity string,
user string,
time string,
request string,
status string,
size string,
referer string,
agent string)
ROW FORMAT SERDE
'org.apache.hadoop.hive.contrib.serde2.RegexSerDe'
WITH SERDEPROPERTIES(
'input.regex'  = '([^ ]*) ([^ ]*) ([^ ]*) (-|\\[[^\\]]*\\]) ([^ \"]*|\"[^\"]*\") (-|[0-9]*) (-|[0-9]*)(?: ([^ \"]*|\"[^\"]*\") ([^ \"]*|\"[^\"]*\"))?',
'output.format.string' = '%1$s %2$s %3$s %4$s %5$s %6$s
%7$s %8$s %9$s');
```

A propriedade input.regex é a expressão regular aplicada em cada registro e o output.format.string indica como os campos de coluna podem ser construídos a partir da verificação das expressões regulares. Este exemplo também ilustra pares de valores chave arbitrários podem ser passados para um “serde” usando a cláusula WITH SERDEPROPERTIES, uma capacidade que pode ser muito útil para passar parâmetros arbitrários para um Serde.

C. Formato de Arquivo

Os arquivos do Hadoop podem ser armazenados em diferentes formatos. Um arquivo no Hadoop especifica como os registros estão armazenados no arquivo. Os arquivos de texto, por exemplo, são armazenados no TextInputFormat e os arquivos binários podem ser armazenados como SequenceFileInputFormat. Usuários também podem implementar seus próprios formatos de arquivo. Hive não impõem restrições quanto ao tipo de formatos de entrada de dados são armazenados. O formato pode ser especificado e criado. Para além dos dois formatos mencionados acima, o Hive também fornece um RCFileInputFormat que armazena os dados em uma coluna de maneira orientada. Tal organização permite importantes melhorias de desempenho especialmente para consultas que não acessam todas as colunas da tabela. Os usuários podem adicionar seus próprios formatos de arquivo e associá-los a uma tabela como mostrado na seguinte declaração.

```
CREATE TABLE dest1(key INT, value STRING)
STORED AS
INPUTFORMAT
'org.apache.hadoop.mapred.SequenceFileInputFormat'
OUTPUTFORMAT
'org.apache.hadoop.mapred.SequenceFileOutputFormat'
```

A cláusula STORED AS especifica as classes a serem usadas para determinar os formatos de entrada e saída dos arquivos na tabela ou o diretório da partição. Esta pode ser qualquer classe que implementa as interfaces java FileInputFormat e FileOutputFormat. As classes podem ser fornecidas ao Hadoop em um jar semelhantes aos mostrados nos exemplos de adição de SerDes personalizado.

IV. Arquitetura do Sistema e Componentes

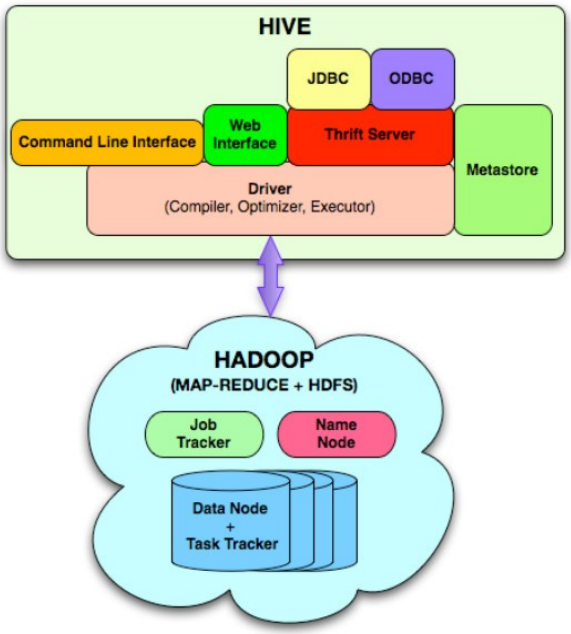

Fig. 1: Arquitetura do Sistema Hive

Os componentes a seguir são os principais blocos do Hive:

• Metastore - O componente que armazena o sistema de catálogo e metadados sobre tabelas, colunas, partições, etc.

• Driver - Componente que gerencia o ciclo de vida de um HiveQL verificando como ele se move através Hive. O driver também mantém um identificador de sessão e algumas sessões estatísticas.

• Query Compiler - O componente que compila o HiveQL em um gráfico acíclico dirigido de tarefas de map / reduce.

• Execution Engine - O componente que executa as tarefas produzidas pelo compilador em orde demdependência apropriada. O mecanismo de execução interage com o Hadoop.

• HiveServer - O componente que fornece uma interface “thrift” e um servidor JDBC / ODBC que permite integrar o Hive com outras aplicações.

• Componentes de clientes com uma interface de linha de comando(CLI), a interface Web e o driver para JDBC/ ODBC.

• Interfaces de Extensibilidade que incluem o SerDe e interfaces ObjectInspector já descritas anteriormente bem como o UDF (User Defined Function) e UDAF (User Defined Aggregate Function) interfaces que permitem aos usuários definir suas próprias funções personalizadas.

Uma instrução HiveQL é enviada via CLI, a interface da web ou um cliente externo usando as interfaces “thrift”, odbc ou jdbc. O driver primeiro passa a consulta para o compilador onde ele vai através da análise típica, verificar o tipo e analisar a semântica utilizada nos metadados armazenados no Metastore. O compilador gera um plano lógico que é então otimizado através de um otimizador baseado em regras simples. Finalmente, um plano na forma de um DAG de tarefas map-reduce e tarefas hdfs é gerado. O mecanismo de execução então executa essas tarefas na ordem de suas dependências, usando o Hadoop. Nesta seção, fornecemos mais detalhes sobre o Metastore, o compilador de consultas e o mecanismo de execução.

A. Metastore

*O Metastore atua como um catálogo do sistema para o Hive. Ele armazena t*odas as informações sobre as tabelas, suas partições, esquemas, colunas e seus tipos, os locais da tabela etc. Estas informações podem ser consultadas ou modificadas usando uma interface thrift([7]) e, como resultado, ele pode ser chamado de clientes em diferentes linguagens de programação. Como esta informação necessita ser servido rapidamente ao compilador, optamos por armazenar estas informações sobre um RDBMS tradicional. O Metastore assim torna-se um aplicativo que é executado em um RDBMS e uma camada ORM open-source chamada DataNucleus ([8]), para converter objeto em um esquema relacional e vice-versa. Optamos por esta abordagem em vez de armazenar esta informação em hdfs como precisamos do Metastore para ter latência muito baixa. A camada de DataNucleus permite muitos nós

plugados no RDBMS. Em nossa implantação no Facebook, usamos Mysql para armazenar essas informações. O Metastore é muito crítico para o Hive. Sem o sistema catálogo não é possível impor uma estrutura de arquivos hadoop.
Como resultado, é importante que as informações do Metastore sejam copiados regularmente. Idealmente um servidor replicado devem também ser implementados para fornecer a disponibilidade que muitos ambientes de produção precisam. Também é importante para garantir que este servidor seja capaz de escalar com o número de consultas enviadas pelos usuários. Hive endereça isso para assegurar que nenhuma chamada Metastore seja feita a partir dos mapeadores ou redutores do job. Todos os metadados necessários ao Mapper ou Reducer é passado através de arquivos de plano xml que são gerados pelo compilador e que contenham qualquer informação necessária em tempo de execução.

A lógica ORM no Metastore pode ser implementada nas Bibliotecas do cliente, de modo que ela seja executada no lado do cliente e chamadas para um RDBMS. Esta implantação é fácil e ideal se os únicos clientes que interagem com o Hive são os CLI ou a interface da web. No entanto, assim que os metadados do Hive necessitem serem manipulados e consultados por programas em linguagens como python, php etc., isto é, por clientes não escritos em java, um servidor Metastore tem de ser implementado.

B. Compilador Query

Os metadados armazenados no Metastore são usados pela query do compilador para gerar o plano de execução. Similar aos compiladores em bancos de dados tradicionais, o compilador Hive processa o HiveQL em instruções nas seguintes etapas:

• Parse - Hive usa Antlr para gerar a sintaxe abstrata de árvore (AST) para a consulta.

• Verificação de Tipo e Análise Semântica - Durante esta fase, o compilador busca as informações de todas as tabelas de entrada e saída do Metastore e usa informações para construir um plano lógico. Verifica a compatibilidade dos tipos nas expressões e sinaliza qualquer erro na fase de compilação. A transformação de um AST para um operador DAG passa por um representação intermediária que é chamada de bloco de consulta (QB). O compilador converte consultas pais e filhas relacionadas em uma árvore QB. Ao mesmo tempo, a representação da árvore QB também ajuda na organização das partes relevantes da árvore AST de forma mais acessível transformando um operador DAG que não seja um AST.

• Otimização - A lógica de otimização consiste em uma cadeia de transformações tais que o operador DAG resultante de uma transformação passada como a próxima transformação. Qualquer pessoa que deseje um compilador ou deseje adicionar nova lógica de otimização pode facilmente fazer isso, implementando a transformação como uma extensão da interface Transform e adicionando-a na cadeia de transformações do otimizador.

A lógica de transformação tipicamente compreende uma caminhada sobre o operador DAG de tal forma que determinadas ações de processamento são tomadas no operador DAG quando as condições relevantes ou regras são satisfeitas. As cinco interfaces principais que são envolvidas na transformação são Node, GraphWalker, Dispatcher, Rule e Processor. Os Nodes do Operador DAG implementam a interface Node que permite que o operador DAG seja manipulado por outras interfaces mencionadas acima. Uma típica transformação envolve caminhar o DAG para cada Node visitado, verificando se uma regra está satisfeita e, em seguida, invocando o processador correspondente para essa regra no caso ser satisfeita mais tarde. O Dispatcher mantém o mapeamento dessas regras para os processadores e e verifica se as regra foram satisfeitas. É passado para o GraphWalker para que o processador apropriado possa ser despachado enquanto está sendo visitado na caminhada. O fluxograma da Fig. 2 mostra como uma transformação típica é estruturada.

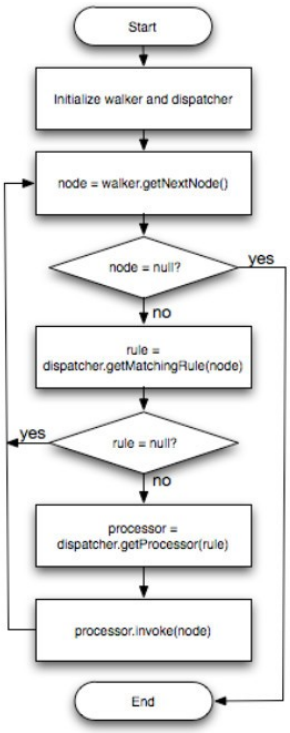

FIG. 2: Fluxograma para transformação típica durante a otimização

As transformações a seguir são feitas atualmente em Hive como parte da fase de otimização:

i. Poda de coluna - Esta etapa de otimização garante que apenas as colunas que são necessários no processamento da consulta são projetadas fora da linha.

ii. Predicado empurrado para baixo - Predicados são empurrados para a varredura, se possível, para que as linhas e filtros estejam no início do processamento.

iii. Poda de partição - Predicados em partição de colunas são usadas para apagar arquivos de partições que não satisfazem o predicado.

iv. Joins do lado do Map - Nos casos em que alguns das tabelas em uma junção são muito pequenas, as tabelas são replicadas em todos os mapeadores e juntando-se com outras tabelas. Esse comportamento é desencadeado por uma dica na consulta desta forma:

```
SELECT /*+ MAPJOIN(t2) */ t1.c1, t2.c1
FROM t1 JOIN t2 ON(t1.c2 = t2.c2);
```

Vários parâmetros controlam a memória que é usada no mapeador para manter o conteúdo da tabela replicada. Estes são hive.mapjoin.size.key e hive.mapjoin.cache.numrows que controlam o número de linhas da tabela que são mantidas na memória a qualquer momento e também fornecem o tamanho da chave de associação.

v. Reordenando os Joins - As tabelas maiores são transmitidas e não são materializadas na memória enquanto o redutor das tabelas menores são mantidas em memória. Isso garante que a operação de junção não exceda os limites de memória do lado do redutor.

Para além do MAPJOIN, o usuário também pode fornecer dicas ou definir parâmetros para fazer o seguinte:

i.Reparticionamento de dados para lidar com processamento GROUP BY - No mundo real muitos conjuntos de dados têm uma distribuição de colunas utilizadas na cláusula GROUP BY para consultas comuns. Nessas situações, o plano habitual de distribuição dos dados no group by de colunas e depois agregando em um redutor que não funciona bem, pois a maioria dos dados são enviados para poucos redutores. Um plano melhor em tais situações seria usar duas etapas de Map / Reduce para computar a agregação. Na primeira fase os dados são distribuídos aleatoriamente (ou distribuídos na coluna DISTINCT em caso de Agregações) aos redutores e às agregações são computadas. Essas agregações são então distribuídos em colunas GROUP BY aos redutores na segunda fase do Map / Reduce. Uma vez que o número de tuplas de agregação parcial é muito menor do que o conjunto de dados de base, isto normalmente leva a um melhor desempenho. Dentro deste comportamento pode ser desencadeado por uma definição de parâmetro da seguinte maneira:

```
set hive.groupby.skewindata=true;

SELECT t1.c1, sum(t1.c2)
FROM t1
GROUP BY t1;
```

ii. As agregações parciais baseadas em hash nos Mapeadores - agregações parciais baseadas em Hash podem potencialmente reduzir os dados que são enviados pelos mapeadores aos redutores. Isto por sua vez reduz a quantidade de tempo gasto na triagem e mesclando esses dados. Como resultado, muitos ganhos de desempenho podem ser alcançados nesta estratégia. O Hive permite aos usuários controlar a quantidade de memória que pode ser usada no mapeador para manter as linhas em uma tabela hash para essa otimização. O parâmetro hive.map.aggr.hash.percentmemory especifica a fração de memória do mapeador que pode ser utilizado para manter a tabela hash, por exemplo, 0,5 asseguraria que assim que o tamanho do hash da tabela exceder a metade da memória para um mapeador, os agregados parciais armazenados nelas são enviados para os redutores.O parâmetro hive.map.aggr.hash.min.reduction também é usado para controlar a quantidade de memória utilizada nos mapeadores.

• Geração do plano físico - O plano lógico gerado no final da fase de otimização é então dividido em múltiplas map / reduce e jobs hdfs. Como um exemplo, um grupo de dados distorcidos pode gerar duas tarefas de map/reduce seguidos por uma tarefa final do hdfs que move os resultados para o local correto no hdfs. No final desta fase, o plano físico parece um DAG de cada tarefa encapsulando uma parte do plano.

Mostramos um exemplo de consulta de inserção de várias planos físicos correspondentes após todas as otimizações abaixo.

FROM (SELECT a.status, b.school, b.gender

FROM status_updates a JOIN profiles b
ON (a.userid = b.userid
AND a.ds='2009-03-20')) subq1

INSERT OVERWRITE TABLE gender_summary
PARTITION(ds='2009-03-20')
SELECT subq1.gender, COUNT(1)
GROUP BY subq1.gender

INSERT OVERWRITE TABLE school_summary
PARTITION(ds='2009-03-20')
SELECT subq1.school, COUNT(1)
GROUP BY subq1.school

Esta consulta tem uma única associação seguida de duas agregações. Ao escrever a consulta como um multi-table-insert, nós nos certificamos que a associação é executada apenas uma vez. O plano para a consulta é mostrada na Fig. 3 abaixo.

Os nós do plano são operadores físicos e as arestas representam o fluxo de dados entre operadores. A última linha em cada nó representa o esquema de saída desse operador. Para falta de espaço, não descrevemos os parâmetros especificados dentro de cada nó de operador. O plano tem três jobs map/reduce.

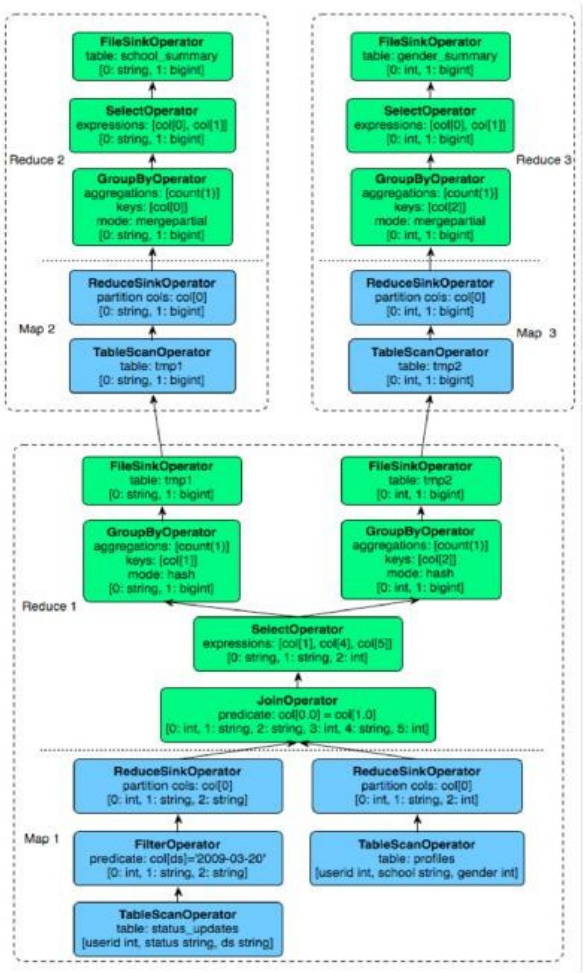

FIG. 3: Plano de consulta para inserção de consultas em múltiplas tabelas com 3 jobs Map/Reduce

Dentro do mesmo job do Map/Reduce, a parte do operador em árvore abaixo do operador de repartição (ReduceSinkOperator) é executada pelo mapeador e a porção acima pelo redutor. O reparticionamento propriamente dito é executado pelo mecanismo de execução.

Observe que o primeiro job Map/Reduce grava em dois arquivos temporários no HDFS, tmp1 e tmp2, que são consumidos pelo segundo e terceiro jobs map-reduces respectivamente. Assim, o segundo e o terceiro param para que o primeiro job de map/reduce termine.

C. Mecanismo de Execução

Finalmente, as tarefas são executadas pela ordem de dependências. Cada tarefa dependente é executada somente se todos os seus pré-requisitos foram executados. Uma tarefa de map/reduce primeiro serializa sua parte do plano em um arquivo plan.xml. Este arquivo é então adicionado ao job de cache para a tarefa e instâncias de ExecMapper e ExecReducers que são gerados usando Hadoop.

Cada uma dessas classes deserializa o plan.xml e executa a parte relevante do operador DAG. Os resultados finais são armazenados em um local temporário. No final da consulta inteira, os dados finais são movidos para o local desejado no caso de LMD. No caso de consultas os dados são servidos como tal no local temporário.

V. Hive usado no Facebook

Hive e Hadoop são usados, extensivamente no Facebook para diferentes tipos de processamento de dados. Atualmente, nosso warehouse tem 700TB de dados (que vem a 2.1PB de espaço bruto em Hadoop após contabilização para a replicação de 3 vias).Nós adicionamos 5TB (15TB após a replicação) de dados comprimidos diariamente. A Taxa de compressão típica é 1:7 e em alguns momentos mais do que isso. Todos os dias mais de 7500 jobs são submetidos ao cluster e mais de 75 TB de dados compactados são processados a cada dia. Com o crescimento contínuo da rede do Facebook, temos o crescimento contínuo dos dados. Ao mesmo tempo que a empresa escala, o cluster também tem que escalar com o crescimento dos usuários.

Mais da metade da carga de trabalho são consultas ad hoc onde, o restate são para relatórios e dashboards. Hive ativou esse tipo de carga de trabalho no cluster do Hadoop no Facebook por causa da simplicidade com que a análise adhoc pode ser feita. Contudo, compartilhando os mesmos recursos com os usuários adhoc e relatórios os usuários apresentam desafios operacionais significativos pela imprevisibilidade dos jobs ad hoc. Muitas vezes esses jobs não são adequadamente ajustado e, portanto, consomem do cluster recursos valiosos. Isso pode, por sua vez, levar a um desempenho nas consultas de relatórios, muitas das quais possuem tempos críticos.

O gerenciamento dos recursos tem sido um tanto fraco no Hadoop e a única solução viável no momento parece estar manter os clusters separados para consultas adhoc e consultas de relatório.

Há também uma grande variedade de jobs do Hive que são executados diariamente. Eles vão desde simples jobs de diferentes tipos de pacotes e cubos para os mais avançados algoritmos de aprendizagem de máquina. O sistema é usado por usuários iniciantes e avançados para que novos usuários usem o sistema imediatamente após uma hora do início de longos treinamentos.

Um resultado de uso pesado também leva a um monte de tabelas geradas no warehouse e isso, por sua vez, aumenta enormemente a necessidade de ferramentas de descoberta de dados, especialmente para novas usabilidades. Em geral, o sistema nos permitiu fornecer serviços de processamento de dados para engenheiros e analistas a uma fração do custo de uma infra-estrutura de armazenagem mais tradicional.

Adicionado, a capacidade do Hadoop para escalar para milhares de nodes dá a confiança de que seremos capazes de escalar esta infra-estrutura ainda mais.

VI. Trabalhos Relacionados

Tem havido uma demanda de trabalho recente sobre escalar petabyte de dados para sistemas de processamento, tanto open-source quanto comercial. Scope[14] é uma linguagem SQL-like que esta no topo do Cosmos da Microsoft map/reduce e sistema de arquivos distribuídos. Pig [13] permite que os usuários escrevam scripts declarativos para processar dados. Hive é diferente desses sistemas, pois fornece um sistema de catálogo que persiste metadados sobre tabelas dentro do sistema. Isto permite que o hive funcione como um warehouse que pode ser interfaceado com ferramentas de relatório padrão como MicroStrategy [16]. HadoopDB [15] reutiliza a maior parte do sistema do Hive, exceto, que usa instâncias de banco de dados tradicionais em cada um dos nós para armazenar dados em vez de usar um sistema de arquivo distribuído.

VII. Conclusões e Trabalhos Futuros

Hive é um trabalho em andamento. É um projeto open-source, que está activamente sendo utilizado pelo Facebook, bem como várias contribuidores externos.

HiveQL atualmente aceita apenas um subconjunto de SQL como querys válidas. Estamos trabalhando para tornar HiveQL uma Sintaxe SQL. O Hive atualmente tem um otimizador baseado em regras ingênuas com um pequeno número de regras simples. Planejamos construir um otimizador e técnicas de otimização adaptativa para vir com planos mais eficientes. Nós estamos explorando armazenamento colunar e colocação de dados mais inteligente para melhorar o desempenho. Estamos executando benchmarks de desempenho baseados em [9] para medir o nosso progresso, bem como comparar com outros sistemas. Em nossos experimentos preliminares,fomos capaz de melhorar o desempenho do próprio Hadoop em cerca de 20% em comparação com [9]. As melhorias envolveram a utilização de estruturas de dados Hadoop para processar os dados, por exemplo, texto em vez de string.

As mesmas queries no HiveQL tiveram um overhead de 20% em relação a nossa implementação do Hadoop, ou seja, o desempenho do Hive está em paridade com o código Hadoop de [9]. Temos também rodado na indústria Benchmark padrões de apoio à decisão - TPC-H [11]. Baseado nessas experiências, identificamos várias áreas para melhor o desempenho e começamos a trabalhar neles. Mais detalhes estão disponíveis em [10] e [12].Estamos aprimorando os drivers JDBC e ODBC para Hive para integração com ferramentas comerciais de BI que funcionam apenas com warehouse. Estamos explorando métodos para ténicas de consultas otimizadas com querys múltiplas e realização de joins n-way genéricos em um único job map/reduce.

RECONHECIMENTO

Gostaríamos de agradecer nossos usuários à comunidade de desenvolvedores por suas contribuições, com um agradecimento especial a Eric Hwang, Yuntao Jia, Yongqiang Ele, Edward Capriolo e Dhruba Borthakur.

REFERÊNCIAS

[1] Apache Hadoop. Available at http://wiki.apache.org/hadoop.

[2] Facebook Lexicon at http://www.facebook.com/lexicon.
[3] Hive wiki at http://www.apache.org/hadoop/hive.
[4] Hadoop Map-Reduce Tutorial at http://hadoop.apache.org/common/docs/current/mapred_tutorial.html.
[5] Hadoop HDFS User Guide at http://hadoop.apache.org/common/docs/current/hdfs_user_guide.html.
[6] Mysql list partitioning at http://dev.mysql.com/doc/refman/5.1/en/partitioning-list.html.
[7] Apache Thrift. Available at http://incubator.apache.org/thrift.
[8] DataNucleus .Available at http://www.datanucleus.org.
[9] A. Pavlo et. al. A Comparison of Approaches to Large-Scale Data
Analysis. In Proc. of ACM SIGMOD, 2009.
[10] Hive Performance Benchmark. Available at http://issues.apache.org/jira/browse/HIVE-396
[11] TPC-H Benchmark. Available at http://www.tpc.org/tpch
[12] Running TPC-H queries on Hive. Available at http://issues.apache.org/jira/browse/HIVE-600
[13] Hadoop Pig. Available at http://hadoop.apache.org/pig
[14] R. Chaiken, et. al. Scope: Easy and Efficient Parallel Processing of
Massive Data Sets. In Proc. of VLDB, 2008.
[15] HadoopDB Project. Available at http://db.cs.yale.edu/hadoopdb/hadoopdb.html
[16] MicroStrategy. Available at http://www.microstrategy.com