Hive 知识体系保姆级总结

本文档来自公众号：**五分钟学大数据**

扫码关注

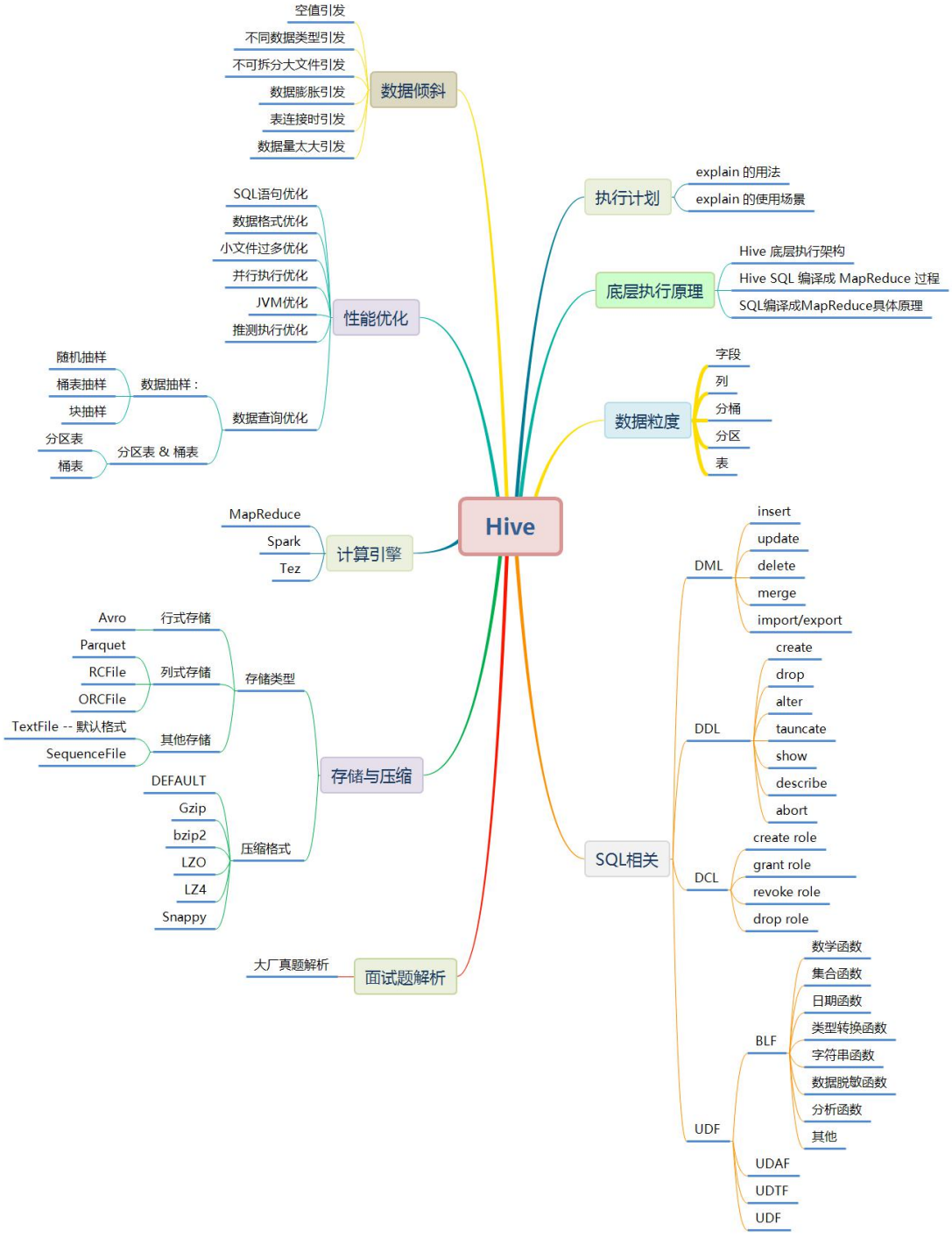
目录

Hive 涉及的知识点如下图所示，本文将逐一讲解：.....5
一. Hive 概览.....6
1.1 hive 的简介.....6
1.2 hive 的架构.....6
1.3 hive 与 hadoop 的关系.....8
1.4 hive 与传统数据库对比.....8
1.5 hive 的数据存储.....9
二、Hive 表类型.....10
2.1 Hive 数据类型.....10
2.2 Hive 内部表.....10
2.3 Hive 外部表.....10
2.4 Hive 分区表.....11
2.5 Hive 分桶表.....11
2.6 Hive 视图.....12
三、Hive 数据抽样.....13
3.1 随机抽样.....13
3.2 块抽样.....13
3.3 桶表抽样.....14
四、Hive 计算引擎.....14
4.1 MR 计算引擎.....14
4.2 Tez 计算引擎.....15
4.3 Spark 计算引擎.....16
五、存储与压缩.....18
5.1 Hive 存储格式.....18
5.2 Hive 压缩格式.....21
5.3 存储和压缩相结合.....23
5.4 主流存储文件性能对比.....24
六、Hive Sql 大全.....26
hive 的 DDL 语法.....27
对数据库的操作.....27
对数据表的操作.....28
hive 的 DQL 查询语法.....35
单表查询.....35
Hive 函数.....38
聚合函数.....38
关系运算.....39
数学运算.....40
逻辑运算.....40
数值运算.....40
条件函数.....42
日期函数.....43

字符串函数 46
复合类型构建操作 52
复杂类型访问操作 52
复杂类型长度统计函数 53
hive 当中的 lateral view 与 explode 以及 reflect 和窗口函数 54
使用 explode 函数将 hive 表中的 Map 和 Array 字段数据进行拆分 54
使用 explode 拆分 json 字符串 56
配合 LATERAL VIEW 使用 58
行转列 58
列转行 60
reflect 函数 61
窗口函数与分析函数 63
sum、avg、min、max 63
row_number、rank、dense_rank、ntile 65
其他一些窗口函数 67
lag,lead,first_value,last_value 67
cume_dist,percent_rank 69
grouping sets,grouping__id,cube,rollup 70
七、Hive 执行计划 73
查看 SQL 的执行计划 74
1. explain 的用法 74
2. explain 的使用场景 77
2. explain dependency 的用法 84
3. explain authorization 的用法 87
最后 88
八、Hive SQL 底层执行原理 88
Hive 底层执行架构 88
Hive SQL 编译成 MapReduce 过程 91
SQL 编译成 MapReduce 具体原理 95
九、Hive 千亿级数据倾斜 97
数据倾斜问题剖析 97
数据倾斜解决方案 97
1. 空值引发的数据倾斜 97
2. 不同数据类型引发的数据倾斜 98
3. 不可拆分大文件引发的数据倾斜 99
4. 数据膨胀引发的数据倾斜 99
5. 表连接时引发的数据倾斜 100
6. 确实无法减少数据量引发的数据倾斜 101
总结 102
十、Hive 企业级性能优化 102
Hive 性能问题排查的方式 102
Hive 性能调优的方式 104
1. SQL 语句优化 104
2. 数据格式优化 106

3. 小文件过多优化.....107
4. 并行执行优化.....107
5. JVM 优化.....108
6. 推测执行优化.....108
十一、Hive 大厂面试真题.....110
1. hive 内部表和外部表的区别.....110
2. Hive 有索引吗.....110
3. 运维如何对 hive 进行调度.....111
4. ORC、Parquet 等列式存储的优点.....111
5. 数据建模用的哪些模型？.....112
6. 为什么要对数据仓库分层？.....115
7. 使用过 Hive 解析 JSON 串吗.....115
8. sort by 和 order by 的区别.....115
9. 数据倾斜怎么解决.....116
10. Hive 小文件过多怎么解决.....116
11. Hive 优化有哪些.....118
附：九个最易出错的 SQL 讲解.....119

Hive 涉及的知识点如下图所示，本文将逐一讲解：



一. Hive 概览

1.1 hive 的简介

Hive 是基于 Hadoop 的一个数据仓库工具，可以将结构化的数据文件映射为一张数据库表，并提供类 SQL 查询功能。

其本质是将 SQL 转换为 MapReduce/Spark 的任务进行运算，底层由 HDFS 来提供数据的存储，说白了 hive 可以理解为一个将 SQL 转换为 MapReduce/Spark 的任务的工具，甚至更进一步可以说 hive 就是一个 MapReduce/Spark Sql 的客户端

为什么要使用 hive ？

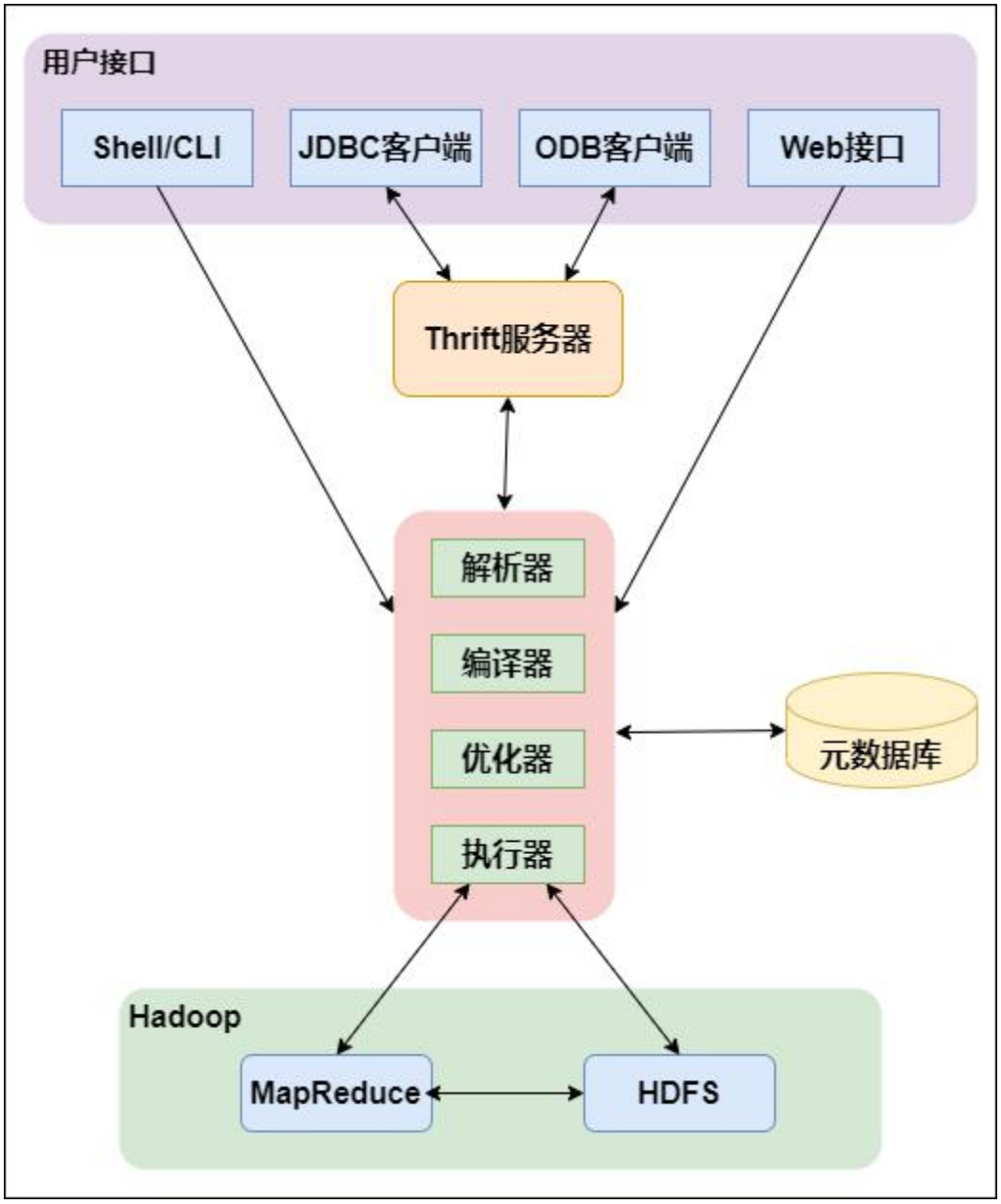
主要的原因有以下几点:

- 学习 MapReduce 的成本比较高，项目周期要求太短，MapReduce 如果要实现复杂的查询逻辑开发的难度是比较大的。
- 而如果使用 hive，hive 采用操作接口类似 SQL 语法，提高快速开发的能力. 避免去书写 MapReduce,减少学习成本，而且提供了功能的扩展

hive 的特点:

1. 可扩展 ：Hive 可以自由的扩展集群的规模，一般情况下不需要重启服务。
2. 延展性 ：Hive 支持用户自定义函数，用户可以根据自己的需求来实现自己的函数。
3. 容错 ： 良好的容错性，节点出现问题 SQL 仍可完成执行。

1.2 hive 的架构



基本组成:

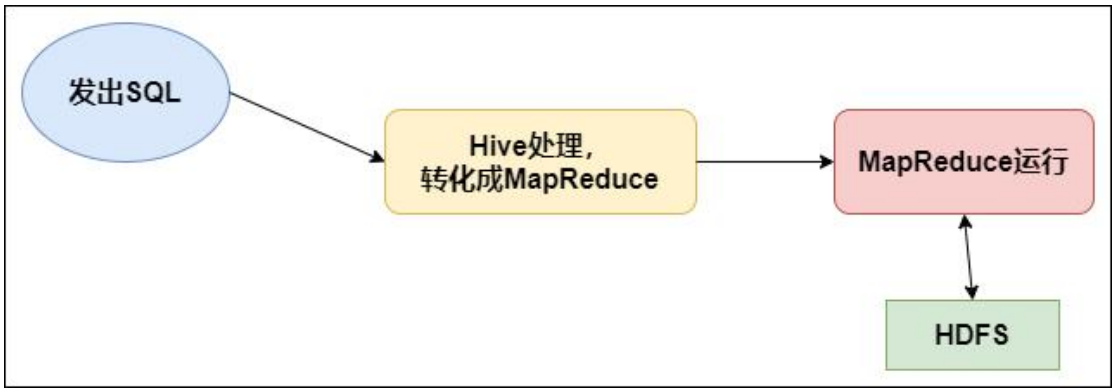
用户接口：包括 CLI、JDBC/ODBC、WebGUI。其中，CLI(command line interface)为 shell 命令行；JDBC/ODBC 是 Hive 的 JAVA 实现，与传统数据库 JDBC 类似；WebGUI 是通过浏览器访问 Hive。

元数据存储：通常是存储在关系数据库如 mysql/derby 中。Hive 将元数据存储在数据库中。Hive 中的元数据包括表的名字，表的列和分区及其属性，表的属性（是否为外部表等），表的数据所在目录等。

解释器、编译器、优化器、执行器:完成 HQL 查询语句从词法分析、语法分析、编译、优化以及查询计划的生成。生成的查询计划存储在 HDFS 中，并在随后有 MapReduce 调用执行。

1.3 hive 与 hadoop 的关系

Hive 利用 HDFS 存储数据，利用 MapReduce 查询分析数据



1.4 hive 与传统数据库对比

hive 主要是用于海量数据的离线数据分析

	Hive	RDBMS
查询语言	HQL	SQL
数据存储	HDFS	Raw Device or Local FS
执行引擎	MapReduce	Excutor
执行延迟	高	低
处理数据规模	大	小
索引	0.8 版本加入位图索引 3.0版本删除索引	有复杂的索引

1. 查询语言。由于 SQL 被广泛的应用在数据仓库中，因此，专门针对 Hive 的特性设计了类 SQL 的查询语言 HQL。熟悉 SQL 开发的开发者可以很方便的使用 Hive 进行开发。
2. 数据存储位置。Hive 是建立在 Hadoop 之上的，所有 Hive 的数据都是存储在 HDFS 中的。而数据库则可以将数据保存在块设备或者本地文件系统中。
3. 数据格式。Hive 中没有定义专门的数据格式，数据格式可以由用户指定，用户定义数据格式需要指定三个属性：列分隔符（通常为空格、”\t”、”\x001″）、行分隔符（”\n”）以及读取文件数据的方

法(Hive 中默认有三个文件格式 TextFile,SequenceFile 以及 RCFile)。由于在加载数据的过程中,不需要从用户数据格式到 Hive 定义的数据格式的转换，因此，Hive 在加载的过程中不会对数据本身进行任何修改，而只是将数据内容复制或者移动到相应的 HDFS 目录中。而在数据库中，不同的数据库有不同的存储引擎，定义了自己的数据格式。所有数据都会按照一定的组织存储，因此，数据库加载数据的过程会比较耗时。

4. 数据更新。由于 Hive 是针对数据仓库应用设计的，而数据仓库的内容是读多写少的。因此，Hive 中不支持对数据的改写和删除，数据是在加载的时候中确定好的。而数据库中的数据通常是需要经常进行修改的，因此可以使用 INSERT INTO ... VALUES 添加数据,使用 UPDATE ... SET 修改数据。
5. 索引。之前已经说过，Hive 在加载数据的过程中不会对数据进行任何处理，甚至不会对数据进行扫描，因此也没有对数据中的某些 Key 建立索引。Hive 要访问数据中满足条件的特定值时，需要暴力扫描整个数据，因此访问延迟较高。由于 MapReduce 的引入， Hive 可以并行访问数据，因此即使没有索引，对于大数据量的访问，Hive 仍然可以体现出优势。数据库中，通常会针对一个或者几个列建立索引，因此对于少量的特定条件的数据的访问，数据库可以有很高的效率，较低的延迟。由于数据的访问延迟较高，决定了 Hive 不适合在线数据查询。
6. 执行。Hive 中大多数查询的执行是通过 Hadoop 提供的 MapReduce 来实现的，而数据库通常有自己的执行引擎。
7. 执行延迟。之前提到，Hive 在查询数据的时候，由于没有索引，需要扫描整个表，因此延迟较高。另外一个导致 Hive 执行延迟高的因素是 MapReduce 框架。由于 MapReduce 本身具有较高的延迟，因此在利用 MapReduce 执行 Hive 查询时，也会有较高的延迟。相对的，数据库的执行延迟较低。当然，这个低是有条件的，即数据规模较小，当数据规模大到超过数据库的处理能力的时候，Hive 的并行计算显然能体现出优势。
8. 可扩展性。由于 Hive 是建立在 Hadoop 之上的，因此 Hive 的可扩展性是和 Hadoop 的可扩展性是一致的（世界上最大的 Hadoop 集群在 Yahoo!，2009 年的规模在 4000 台节点左右）。而数据库由于 ACID 语义的严格限制，扩展行非常有限。目前最先进的并行数据库 Oracle 在理论上的扩展能力也只有 100 台左右。
9. 数据规模。由于 Hive 建立在集群上并可以利用 MapReduce 进行并行计算，因此可以支持很大规模的数据；对应的，数据库可以支持的数据规模较小。

总结：hive 具有 sql 数据库的外表，但应用场景完全不同，hive 只适合用来做批量数据统计分析。

1.5 hive 的数据存储

1. Hive 中所有的数据都存储在 HDFS 中，没有专门的数据存储格式（可支持 Text，SequenceFile，ParquetFile，ORC 格式 RCFILE 等）

SequenceFile 是 hadoop 中的一种文件格式： 文件内容是以序列化的 kv 对象来组织的

2. 只需要在创建表的时候告诉 Hive 数据中的列分隔符和行分隔符，Hive 就可以解析数据。
3. Hive 中包含以下数据模型：DB、Table，External Table，Partition，Bucket。

- db：在 hdfs 中表现为 hive.metastore.warehouse.dir 目录下一个文件夹。
- table：在 hdfs 中表现所属 db 目录下一个文件夹。
- external table：与 table 类似，不过其数据存放位置可以在任意指定路径。
- partition：在 hdfs 中表现为 table 目录下的子目录。
- bucket：在 hdfs 中表现为同一个表目录下根据 hash 散列之后的多个文件。

二、Hive 表类型

2.1 Hive 数据类型

Hive 的基本数据类型有：TINYINT，SAMLLINT，INT，BIGINT，BOOLEAN，FLOAT，DOUBLE，STRING，TIMESTAMP(V0.8.0+)和 BINARY(V0.8.0+)。

Hive 的集合类型有：STRUCT，MAP 和 ARRAY。

Hive 主要有四种数据模型(即表)：内部表、外部表、分区表和桶表。

表的元数据保存传统的数据库的表中，当前 hive 只支持 Derby 和 MySQL 数据库。

2.2 Hive 内部表

Hive 中的内部表和传统数据库中的表在概念上是类似的，Hive 的每个表都有自己的存储目录，除了外部表外，所有的表数据都存放在配置在 hive-site.xml 文件的${hive.metastore.warehouse.dir}/table_name 目录下。

创建内部表：

```
CREATE TABLE IF NOT EXISTS students(user_no INT,name STRING,sex STRING,
        grade STRING COMMOT '班级'）COMMONT '学生表'
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
STORE AS TEXTFILE;
```

2.3 Hive 外部表

被 external 修饰的为外部表（external table），外部表指向已经存在在 Hadoop HDFS 上的数据，除了在删除外部表时只删除元数据而不会删除表数据外，其他和内部表很像。

创建外部表：

```
CREATE EXTERNAL TABLE IF NOT EXISTS students(user_no INT,name STRING,sex STRING,
        class STRING COMMOT '班级'）COMMONT '学生表'
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
STORE AS SEQUENCEFILE
LOCATION '/usr/test/data/students.txt';
```

2.4 Hive 分区表

分区表的每一个分区都对应数据库中相应分区列的一个索引，但是其组织方式和传统的关系型数据库不同。在 Hive 中，分区表的每一个分区都对应表下的一个目录，所有的分区的数据都存储在对应的目录中。

比如说，分区表 partitinTable 有包含 nation(国家)、ds(日期)和 city(城市)3 个分区，其中 nation = china，ds = 20130506，city = Shanghai 则对应 HDFS 上的目录为：

/datawarehouse/partitinTable/nation=china/city=Shanghai/ds=20130506/。

分区中定义的变量名不能和表中的列相同。

创建分区表：

```
CREATE TABLE IF NOT EXISTS students(user_no INT,name STRING,sex STRING,
        class STRING COMMOT '班级'）COMMONT '学生表'
PARTITIONED BY (ds STRING,country STRING)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
STORE AS SEQUENCEFILE;
```

2.5 Hive 分桶表

桶表就是对指定列进行哈希(hash)计算，然后会根据 hash 值进行切分数据，将具有不同 hash 值的数据写到每个桶对应的文件中。

将数据按照指定的字段进行分成多个桶中去，说白了就是将数据按照字段进行划分，可以将数据按照字段划分到多个文件当中去。

创建分桶表：

```
CREATE TABLE IF NOT EXISTS students(user_no INT,name STRING,sex STRING,
        class STRING COMMOT '班级',score SMALLINT COMMOT '总分'）COMMONT '学生表'
PARTITIONED BY (ds STRING,country STRING)
CLUSTERED BY(user_no) SORTED BY(score) INTO 32 BUCKETS
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
STORE AS SEQUENCEFILE;
```

2.6 Hive 视图

在 Hive 中，视图是逻辑数据结构，可以通过隐藏复杂数据操作（Joins，子查询，过滤,数据扁平化）来于简化查询操作。

与关系数据库不同的是，Hive 视图并不存储数据或者实例化。一旦创建 HIve 视图，它的 schema 也会立刻确定下来。对底层表后续的更改(如 增加新列)并不会影响视图的 schema。如果底层表被删除或者改变，之后对视图的查询将会 failed。基于以上 Hive view 的特性，我们在 ETL 和数据仓库中**对于经常变化的表应慎重使用视图**。

创建视图：

```
CREATE VIEW employee_skills
 AS
SELECT name, skills_score['DB'] AS DB,
skills_score['Perl'] AS Perl,
skills_score['Python'] AS Python,
skills_score['Sales'] as Sales,
skills_score['HR'] as HR
FROM employee;
```

创建视图的时候是不会触发 MapReduce 的 Job，因为只存在元数据的改变。但是，当对视图进行查询的时候依然会触发一个 MapReduce Job 进程：SHOW CREATE TABLE 或者 DESC FORMATTED TABLE 语句来显示通过 CREATE VIEW 语句创建的视图。以下是对 Hive 视图的 DDL 操作：

更改视图的属性：

```
ALTER VIEW employee_skills
SET TBLPROPERTIES ('comment' = 'This is a view');
```

重新定义视图：

```
ALTER VIEW employee_skills AS
SELECT * from employee ;
```

删除视图：

```
DROP VIEW employee_skills;
```

三、Hive 数据抽样

当数据规模不断膨胀时，我们需要找到一个数据的子集来加快数据分析效率。因此我们就需要通过筛选和分析数据集为了进行**模式 & 趋势识别**。目前来说有三种方式来进行抽样：随机抽样，桶表抽样，和块抽样。

3.1 随机抽样

关键词：rand()函数。

使用 rand()函数进行随机抽样，limit 关键字限制抽样返回的数据，其中 rand 函数前的 distribute 和 sort 关键字可以保证数据在 mapper 和 reducer 阶段是随机分布的。

案例如下：

```
select * from table_name
where col=xxx
distribute by rand() sort by rand()
limit num;
```

使用 order 关键词:

案例如下：

```
select * from table_name
where col=xxx
order by rand()
limit num;
```

经测试对比，千万级数据中进行随机抽样 order by 方式耗时更长，大约多 30 秒左右。

3.2 块抽样

关键词：tablesample()函数。

1. tablesample(n percent) 根据 hive 表数据的大小按比例抽取数据，并保存到新的 hive 表中。如：抽取原 hive 表中 10%的数据

注意：测试过程中发现，select 语句不能带 where 条件且不支持子查询，可通过新建中间表或使用随机抽样解决。

```
select * from xxx tablesample(10 percent) 数字与 percent 之间要有空格
```

2. tablesample(nM) 指定抽样数据的大小，单位为 M。

```
select * from xxx tablesample(20M) 数字与 M 之间不要有空格
```

3. tablesample(n rows) 指定抽样数据的行数，其中 n 代表每个 map 任务均取 n 行数据，map 数量可通过 hive 表的简单查询语句确认（关键词：number of mappers: x）

```
select * from xxx tablesample(100 rows) 数字与 rows 之间要有空格
```

3.3 桶表抽样

关键词：tablesample (bucket x out of y [on colname])。

其中 x 是要抽样的桶编号，桶编号从 1 开始，colname 表示抽样的列，y 表示桶的数量。

hive 中分桶其实就是根据某一个字段 Hash 取模，放入指定数据的桶中，比如将表 table_1 按照 ID 分成 100 个桶，其算法是 hash(id) % 100，这样，hash(id) % 100 = 0 的数据被放到第一个桶中，hash(id) % 100 = 1 的记录被放到第二个桶中。创建分桶表的关键语句为：CLUSTER BY 语句。

例如：将表随机分成 10 组，抽取其中的第一个桶的数据：

```
select * from table_01
tablesample(bucket 1 out of 10 on rand())
```

四、Hive 计算引擎

目前 Hive 支持 MapReduce、Tez 和 Spark 三种计算引擎。

4.1 MR 计算引擎

MR 运行的完整过程：

Map 在读取数据时，先将数据拆分成若干数据，并读取到 Map 方法中被处理。数据在输出的时候，被分成若干分区并写入内存缓存（buffer）中，内存缓存被数据填充到一定程度会溢出到磁盘并排序，当 Map 执行完后会将一个机器上输出的临时文件进行归并存入到 HDFS 中。

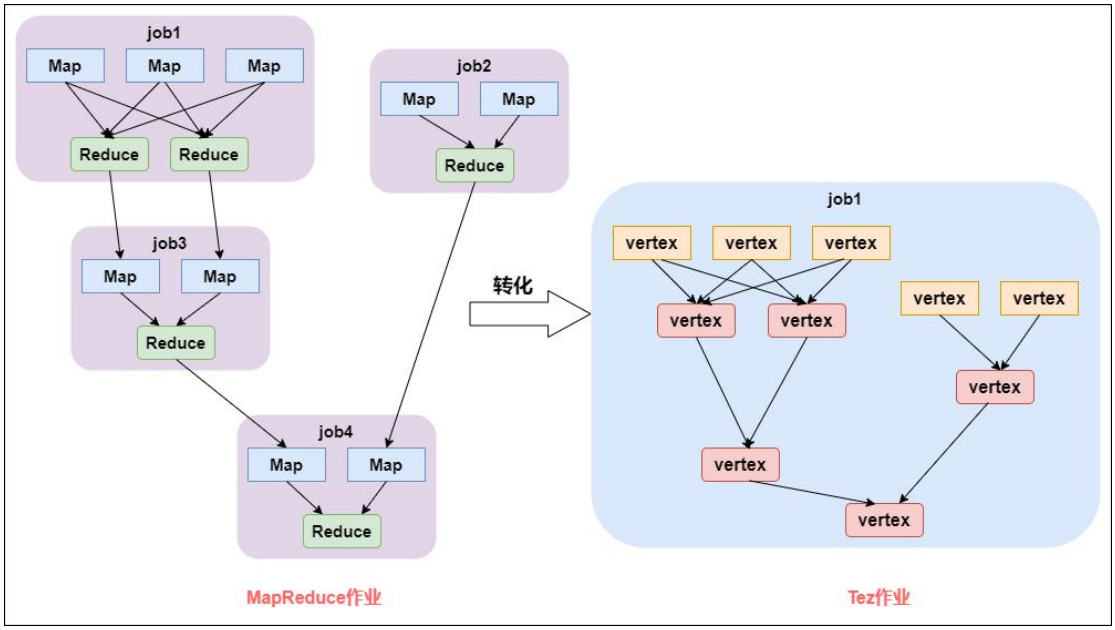
当 Reduce 启动时，会启动一个线程去读取 Map 输出的数据，并写入到启动 Reduce 机器的内存中，在数据溢出到磁盘时会对数据进行再次排序。当读取数据完成后会将临时文件进行合并，作为 Reduce 函数的数据源。

4.2 Tez 计算引擎

Apache Tez 是进行大规模数据处理且支持 DAG 作业的计算框架，它直接源于 MapReduce 框架，除了能够支持 MapReduce 特性，还支持新的作业形式，并允许不同类型的作业能够在一个集群中运行。

Tez 将原有的 Map 和 Reduce 两个操作简化为一个概念——Vertex，并将原有的计算处理节点拆分成多个组成部分：Vertex Input、Vertex Output、Sorting、Shuffling 和 Merging。计算节点之间的数据通信被统称为 Edge，这些分解后的元操作可以任意灵活组合，产生新的操作，这些操作经过一些控制程序组装后，可形成一个大的 DAG 作业。

通过允许 Apache Hive 运行复杂的 DAG 任务，Tez 可以用来处理数据，之前需要多个 MR jobs，现在一个 Tez 任务中。



Tez 和 MapReduce 作业的比较：

- Tez 绕过了 MapReduce 很多不必要的中间的数据存储和读取的过程，直接在一个作业中表达了 MapReduce 需要多个作业共同协作才能完成的事情。
- Tez 和 MapReduce 一样都运行使用 YARN 作为资源调度和管理。但与 MapReduce on YARN 不同，Tez on YARN 并不是将作业提交到

ResourceManager，而是提交到 AMPoolServer 的服务上，AMPoolServer 存放着若干已经预先启动 ApplicationMaster 的服务。

- 当用户提交一个作业上来后，AMPoolServer 从中选择一个 ApplicationMaster 用于管理用户提交上来的作业，这样既可以节省 ResourceManager 创建 ApplicationMaster 的时间，而又能够重用每个 ApplicationMaster 的资源，节省了资源释放和创建时间。

Tez 相比于 MapReduce 有几点重大改进：

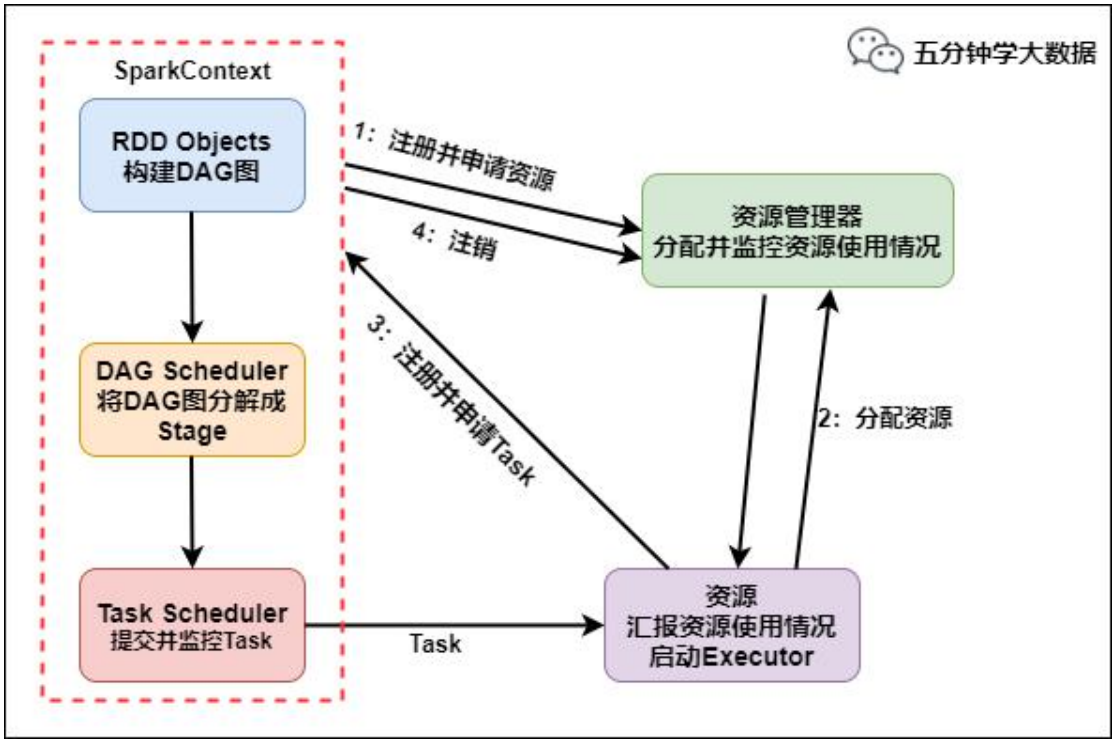
- 当查询需要有多个 reduce 逻辑时,Hive 的 MapReduce 引擎会将计划分解，每个 Redcue 提交一个 MR 作业。这个链中的所有 MR 作业都需要逐个调度，每个作业都必须从 HDFS 中重新读取上一个作业的输出并重新洗牌。而在 Tez 中，几个 reduce 接收器可以直接连接，数据可以流水线传输，而不需要临时 HDFS 文件，这种模式称为 MRR（Map-reduce-reduce*）。
- Tez 还允许一次发送整个查询计划，实现应用程序动态规划，从而使框架能够更智能地分配资源，并通过各个阶段流水线传输数据。对于更复杂的查询来说，这是一个巨大的改进，因为它消除了 IO/sync 障碍和各个阶段之间的调度开销。
- 在 MapReduce 计算引擎中，无论数据大小，在洗牌阶段都以相同的方式执行，将数据序列化到磁盘，再由下游的程序去拉取，并反序列化。Tez 可以允许小数据集完全在内存中处理，而 MapReduce 中没有这样的优化。仓库查询经常需要在处理完大量的数据后对小型数据集进行排序或聚合，Tez 的优化也能极大地提升效率。

4.3 Spark 计算引擎

Apache Spark 是专为大规模数据处理而设计的快速、通用支持 DAG（有向无环图）作业的计算引擎，类似于 Hadoop MapReduce 的通用并行框架，可用来构建大型的、低延迟的数据分析应用程序。

Spark 是用于大规模数据处理的统一分析引擎，基于内存计算，提高了在大数据环境下数据处理的实时性，同时保证了高容错性和高可伸缩性，允许用户将 Spark 部署在大量硬件之上，形成集群。

Spark 运行流程



Spark 运行流程

Spark 具有以下几个特性。

1. **高效性**

Spark 会将作业构成一个 DAG，优化了大型作业一些重复且浪费资源的操作，对查询进行了优化，重新编写了物理执行引擎，如可以实现 MRR 模式。

2. **易用性**

Spark 不同于 MapReducer 只提供两种简单的编程接口，它提供了多种编程接口去操作数据，这些操作接口如果使用 MapReduce 去实现，需要更多的代码。Spark 的操作接口可以分为两类：transformation（转换）和 action（执行）。Transformation 包含 map、flatmap、distinct、reduceByKey 和 join 等转换操作；Action 包含 reduce、collect、count 和 first 等操作。

3. **通用性**

Spark 针对实时计算、批处理、交互式查询，提供了统一的解决方案。但在批处理方面相比于 MapReduce 处理同样的数据，Spark 所要求的硬件设施更高，MapReduce 在相同的设备下所能处理的数据量会比 Spark 多。所以在实际工作中，Spark 在批处理方面只能算是 MapReduce 的一种补充。

4. **兼容性**

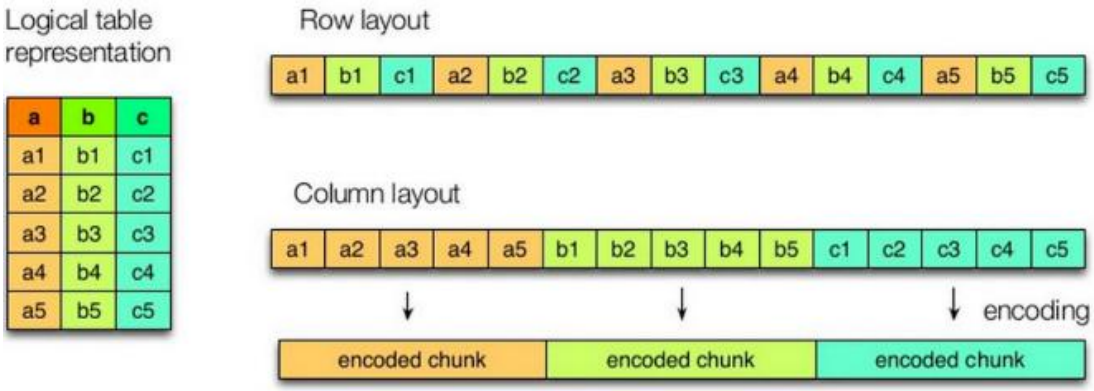
Spark 和 MapReduce 一样有丰富的产品生态做支撑。例如 Spark 可以使用 YARN 作为资源管理器，Spark 也可以处理 Hbase 和 HDFS 上的数据。

五、存储与压缩

5.1 Hive 存储格式

Hive 支持的存储数的格式主要有：TEXTFILE（行式存储） 、SEQUENCEFILE(行式存储)、ORC（列式存储）、PARQUET（列式存储）。

5.1.1 行式存储和列式存储



上图左边为逻辑表，右边第一个为行式存储，第二个为列式存储。

行存储的特点： 查询满足条件的一整行数据的时候，列存储则需要去每个聚集的字段找到对应的每个列的值，行存储只需要找到其中一个值，其余的值都在相邻地方，所以此时行存储查询的速度更快。select *

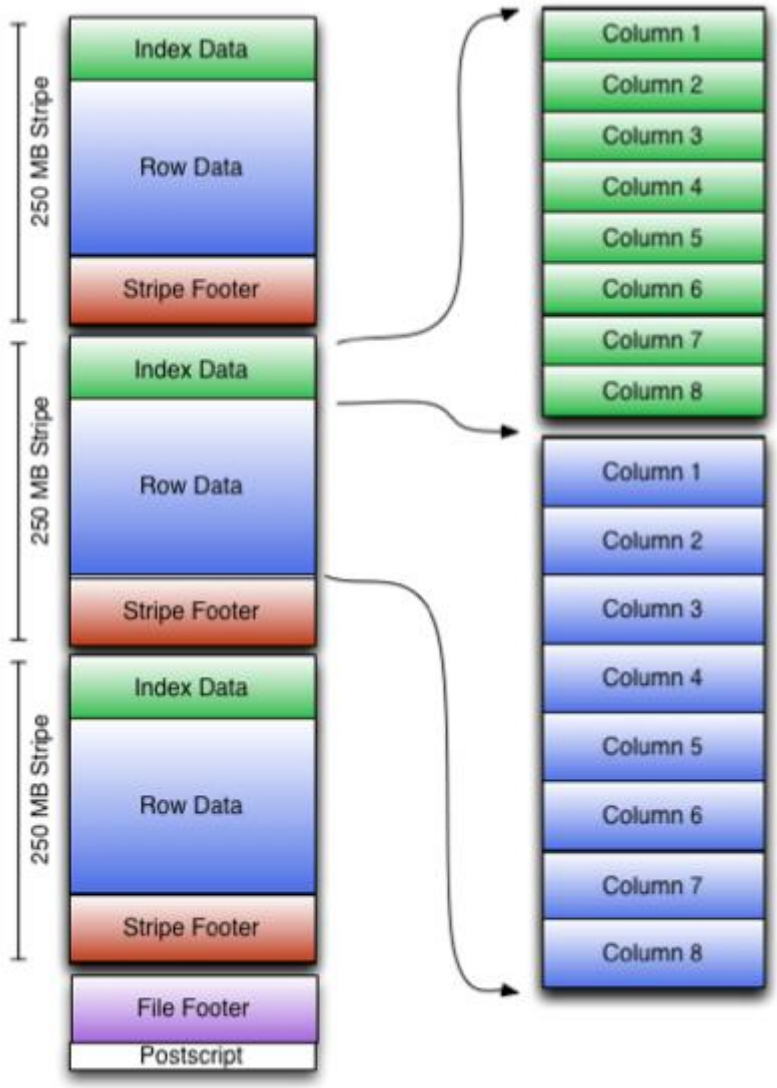
列存储的特点： 因为每个字段的数据聚集存储，在查询只需要少数几个字段的时候，能大大减少读取的数据量；每个字段的数据类型一定是相同的，列式存储可以针对性的设计更好的设计压缩算法。 select 某些字段效率更高。

5.1.2 TEXTFILE

默认格式，数据不做压缩，磁盘开销大，数据解析开销大。可结合 Gzip、Bzip2 使用(系统自动检查，执行查询时自动解压)，但使用这种方式，hive 不会对数据进行切分，从而无法对数据进行并行操作。

5.1.3 ORC 格式

Orc (Optimized Row Columnar)是 hive 0.11 版里引入的新的存储格式。
可以看到每个 Orc 文件由 1 个或多个 stripe 组成，每个 stripe250MB 大小，这个 Stripe 实际相当于 RowGroup 概念，不过大小由 4MB->250MB，这样能提升顺序读的吞吐率。每个 Stripe 里有三部分组成，分别是 Index Data,Row Data,Stripe Footer：



1. Index Data：一个轻量级的 index，默认是每隔 1W 行做一个索引。这里做的索引只是记录某行的各字段在 Row Data 中的 offset。
2. Row Data：存的是具体的数据，先取部分行，然后对这些行按列进行存储。对每个列进行了编码，分成多个 Stream 来存储。
3. Stripe Footer：存的是各个 stripe 的元数据信息

每个文件有一个 File Footer，这里面存的是每个 Stripe 的行数，每个 Column 的数据类型信息等；每个文件的尾部是一个 PostScript，这里面记录了整个文件的压缩类型以及 FileFooter 的长度信息等。在读取文件时，会 seek 到文件尾

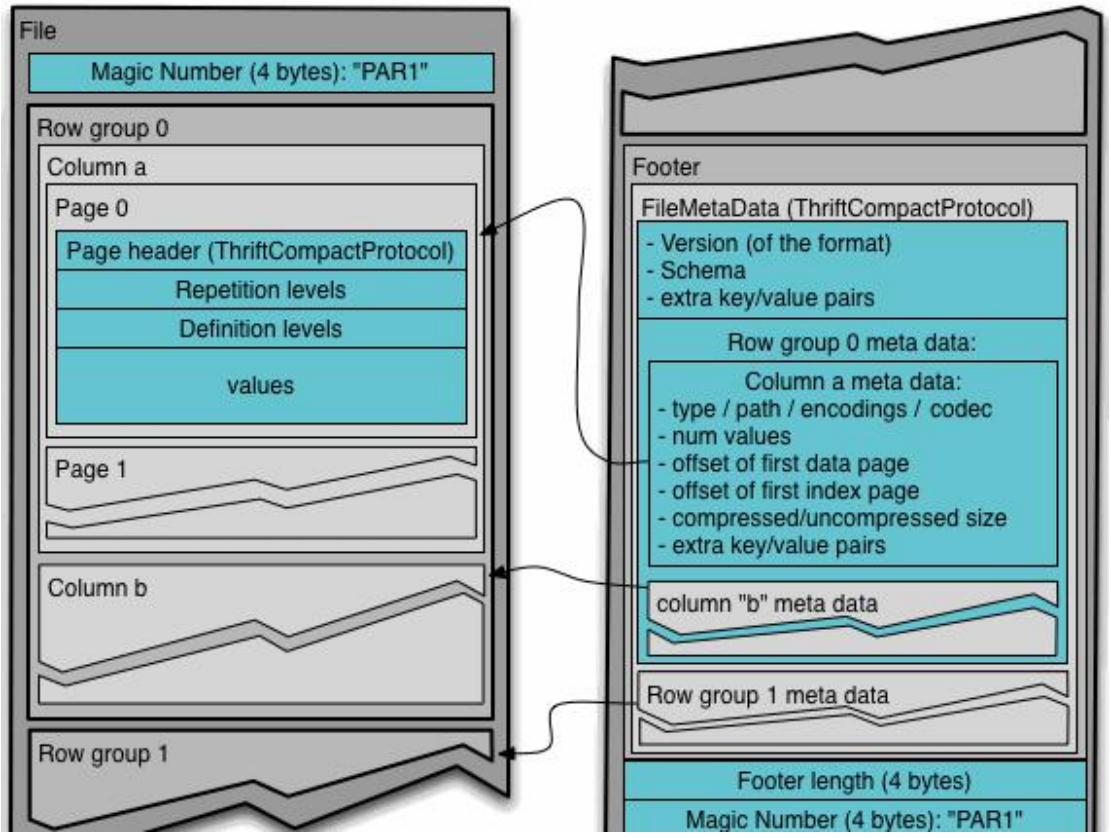
部读 PostScript，从里面解析到 File Footer 长度，再读 FileFooter，从里面解析到各个 Stripe 信息，再读各个 Stripe，即从后往前读。

5.1.4 PARQUET 格式

Parquet 是面向分析型业务的列式存储格式，由 Twitter 和 Cloudera 合作开发，2015 年 5 月从 Apache 的孵化器里毕业成为 Apache 顶级项目。

Parquet 文件是以二进制方式存储的，所以是不可以直接读取的，文件中包括该文件的数据和元数据，因此 Parquet 格式文件是自解析的。

通常情况下，在存储 Parquet 数据的时候会按照 Block 大小设置行组的大小，由于一般情况下每一个 Mapper 任务处理数据的最小单位是一个 Block，这样可以把每一个行组由一个 Mapper 任务处理，增大任务执行并行度。Parquet 文件的格式如下图所示。



上图展示了一个 Parquet 文件的内容，一个文件中可以存储多个行组，文件的首位都是该文件的 Magic Code，用于校验它是否是一个 Parquet 文件，Footer length 记录了文件元数据的大小，通过该值和文件长度可以计算出元数据的偏移量，文件的元数据中包括每一个行组的元数据信息和该文件存储数据的 Schema 信息。除了文件中每一个行组的元数据，每一页的开始都会存储该页的

元数据，在 Parquet 中，有三种类型的页：数据页、字典页和索引页。数据页用于存储当前行组中该列的值，字典页存储该列值的编码字典，每一个列块中最多包含一个字典页，索引页用来存储当前行组下该列的索引，目前 Parquet 中还不支持索引页。

5.2 Hive 压缩格式

在实际工作当中，hive 当中处理的数据，一般都需要经过压缩，前期我们在学习 hadoop 的时候，已经配置过 hadoop 的压缩，我们这里的 hive 也是一样的可以使用压缩来节省我们的 MR 处理的网络带宽

mr 支持的压缩格式:

压缩格式	工具	算法	文件扩展名	是否可切分
DEFAULT	无	DEFAULT	.deflate	否
Gzip	gzip	DEFAULT	.gz	否
bzip2	bzip2	bzip2	.bz2	是
LZO	lzop	LZO	.lzo	是(需手动加索引)
LZ4	无	LZ4	.lz4	否
Snappy	无	Snappy	.snappy	否

hadoop 支持的解压缩的类:

压缩格式	对应的编码/解码器
DEFLATE	org.apache.hadoop.io.compress.DefaultCodec
gzip	org.apache.hadoop.io.compress.GzipCodec
bzip2	org.apache.hadoop.io.compress.BZip2Codec
LZO	com.hadoop.compression.lzo.LzopCodec
LZ4	org.apache.hadoop.io.compress.Lz4Codec
Snappy	org.apache.hadoop.io.compress.SnappyCodec

压缩性能的比较:

压缩算法	原始文件大小	压缩文件大小	压缩速度	解压速度
gzip	8.3GB	1.8GB	17.5MB/s	58MB/s

压缩算法	原始文件大小	压缩文件大小	压缩速度	解压速度
bzip2	8.3GB	1.1GB	2.4MB/s	9.5MB/s
LZO	8.3GB	2.9GB	49.3MB/s	74.6MB/s

Snappy 生成的压缩文件要大 20%到 100%。在 64 位模式下的 core i7 处理器的单内核上，Snappy 以 250 MB/秒或更多的速度压缩，并以 500 MB/秒或更多的速度解压。

实现压缩 hadoop 需要配置的压缩参数:

参数	默认值	阶段	建议
io.compression.codecs（在 core-site.xml 中配置）	org.apache.hadoop.io.compress.DefaultCodec, org.apache.hadoop.io.compress.GzipCodec, org.apache.hadoop.io.compress.BZip2Codec, org.apache.hadoop.io.compress.Lz4Codec	输入压缩	Hadoop 使用文件扩展名判断是否支持某种编解码器
mapreduce.map.output.compress	false	mapper 输出	这个参数设为 true 启用压缩
mapreduce.map.output.compress.codec	org.apache.hadoop.io.compress.DefaultCodec	mapper 输出	使用 LZO、LZ4 或 snappy 编解码器在此阶段压缩数据
mapreduce.output.fileoutputformat.compress	false	reducer 输出	这个参数设为 true 启用压缩
mapreduce.output.fileoutputformat.compress.codec	org.apache.hadoop.io.compress.DefaultCodec	reducer 输出	使用标准工具或者编解码器，如 gzip 和 bzip2
mapreduce.output.fileoutputformat.compress.type	RECORD	reducer 输出	SequenceFile 输出使用的压缩类型：NONE 和 BLOCK

hive 配置压缩的方式:

1. 开启 map 端的压缩方式:

```
1.1）开启 hive 中间传输数据压缩功能
 hive (default)>set hive.exec.compress.intermediate=true;
1.2）开启 mapreduce 中 map 输出压缩功能
 hive (default)>set mapreduce.map.output.compress=true;
1.3）设置 mapreduce 中 map 输出数据的压缩方式
 hive (default)>set mapreduce.map.output.compress.codec= org.apache.hadoop.io.compress.SnappyCodec;
1.4）执行查询语句
 select count(1) from score;
```

2. 开启 reduce 端的压缩方式

```
1）开启 hive 最终输出数据压缩功能
 hive (default)>set hive.exec.compress.output=true;
2）开启 mapreduce 最终输出数据压缩
 hive (default)>set mapreduce.output.fileoutputformat.compress=true;
3）设置 mapreduce 最终数据输出压缩方式
 hive (default)> set mapreduce.output.fileoutputformat.compress.codec = org.apache.
hadoop.io.compress.SnappyCodec;
4）设置 mapreduce 最终数据输出压缩为块压缩
 hive (default)>set mapreduce.output.fileoutputformat.compress.type=BLOCK;
5）测试一下输出结果是否是压缩文件
 insert overwrite local directory '/export/servers/snappy' select * from score dist
ribute by s_id sort by s_id desc;
```

5.3 存储和压缩相结合

ORC 存储方式的压缩：

Key	Default	Notes
orc.compress	ZLIB	高级压缩(可选: NONE, ZLIB, SNAPPY)
orc.compress.size	262,144	每个压缩块中的字节数
orc.stripe.size	67,108,864	每条 stripe 中的字节数
orc.row.index.stride	10,000	索引条目之间的行数(必须是>= 1000)
orc.create.index	true	是否创建行索引
orc.bloom.filter.columns	""	逗号分隔的列名列表，应该为其创建 bloom 过滤器
orc.bloom.filter.fpp	0.05	bloom 过滤器的假阳性概率(必须是>0.0 和 <1.0)

创建一个非压缩的 ORC 存储方式：

```
1）建表语句
    create table log_orc_none(
    track_time string,
    url string,
    session_id string,
    referer string,
    ip string,
    end_user_id string,
    city_id string
```

```
    )ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t' STORED AS orc tblproperties ("o
rc.compress"="NONE");
2）插入数据
 insert into table log_orc_none select * from log_text ;
3）查看插入后数据
 dfs -du -h /user/hive/warehouse/myhive.db/log_orc_none;
 结果显示:
 7.7 M  /user/hive/warehouse/log_orc_none/123456_0
```

创建一个 SNAPPY 压缩的 ORC 存储方式：

```
1）建表语句
    create table log_orc_snappy(
    track_time string,
    url string,
    session_id string,
    referer string,
    ip string,
    end_user_id string,
    city_id string
    )ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t' STORED AS orc tblproperties ("o
rc.compress"="SNAPPY");
2）插入数据
 insert into table log_orc_snappy select * from log_text ;
3）查看插入后数据
 dfs -du -h /user/hive/warehouse/myhive.db/log_orc_snappy ;
 结果显示:
 3.8 M  /user/hive/warehouse/log_orc_snappy/123456_0
4）上一节中默认创建的 ORC 存储方式，导入数据后的大小为
 2.8 M  /user/hive/warehouse/log_orc/123456_0
 比 Snappy 压缩的还小。原因是 orc 存储文件默认采用 ZLIB 压缩。比 snappy 压缩的小。
5）存储方式和压缩总结：
 在实际的项目开发当中，hive 表的数据存储格式一般选择：orc 或 parquet。压缩方式一般选择 snappy。
```

5.4 主流存储文件性能对比

从存储文件的压缩比和查询速度两个角度对比。

压缩比比较：

- TextFile

```
（1）创建表，存储数据格式为 TEXTFILE
    create table log_text (
    track_time string,
    url string,
```

```
    session_id string,
    referer string,
    ip string,
    end_user_id string,
    city_id string
    )ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t' STORED AS TEXTFILE ;
```

```
（2）向表中加载数据
load data local inpath '/export/servers/hivedatas/log.data' into table log_text ;
```

```
（3）查看表中数据大小，大小为 18.1M
dfs -du -h /user/hive/warehouse/myhive.db/log_text;
结果显示：
18.1 M  /user/hive/warehouse/log_text/log.data
```

- ORC

```
（1）创建表，存储数据格式为 ORC
    create table log_orc(
    track_time string,
    url string,
    session_id string,
    referer string,
    ip string,
    end_user_id string,
    city_id string
    )ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t' STORED AS orc ;
```

```
（2）向表中加载数据
insert into table log_orc select * from log_text ;
```

```
（3）查看表中数据大小
dfs -du -h /user/hive/warehouse/myhive.db/log_orc;
结果显示：
2.8 M  /user/hive/warehouse/log_orc/123456_0
```

- Parquet

```
1）创建表，存储数据格式为 parquet
    create table log_parquet(
    track_time string,
    url string,
    session_id string,
    referer string,
    ip string,
```

```
    end_user_id string,
    city_id string
    )ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t' STORED AS PARQUET ;
```

```
2）向表中加载数据
 insert into table log_parquet select * from log_text ;
```

```
3）查看表中数据大小
 dfs -du -h /user/hive/warehouse/myhive.db/log_parquet;
 结果显示:
 13.1 M  /user/hive/warehouse/log_parquet/123456_0
```

数据压缩比结论:

ORC > Parquet > textFile

存储文件的查询效率测试

- textFile

```
hive (default)> select count(*) from log_text;
_c0
100000
Time taken: 21.54 seconds, Fetched: 1 row(s)
```

- ORC

```
hive (default)> select count(*) from log_orc;
_c0
100000
Time taken: 20.867 seconds, Fetched: 1 row(s)
```

- Parquet

```
hive (default)> select count(*) from log_parquet;
_c0
100000
Time taken: 22.922 seconds, Fetched: 1 row(s)
```

存储文件的查询效率比较:

ORC > TextFile > Parquet

六、Hive Sql 大全

本节基本涵盖了 Hive 日常使用的所有 SQL，因为 SQL 太多，所以将 SQL 进行了如下分类： 一、DDL 语句（数据定义语句）：

对数据库的操作：包含创建、修改数据库
对数据表的操作：分为内部表及外部表，分区表和分桶表
二、DQL 语句（数据查询语句）：
单表查询、关联查询
hive 函数：包含聚合函数，条件函数，日期函数，字符串函数等
行转列及列转行：lateral view 与 explode 以及 reflect
窗口函数与分析函数
其他一些窗口函数

hive 的 DDL 语法

对数据库的操作

- 创建数据库:

```
create database if not exists myhive;
说明：hive 的表存放位置模式是由 hive-site.xml 当中的一个属性指定
的 :hive.metastore.warehouse.dir

创建数据库并指定 hdfs 存储位置 :
create database myhive2 location '/myhive2';
```

- 修改数据库:

```
alter  database  myhive2  set  dbproperties('createtime'='20210329');
```

说明：可以使用 alter database 命令来修改数据库的一些属性。但是数据库的元数据信息是不可更改的，包括数据库的名称以及数据库所在的位置

- 查看数据库详细信息

```
查看数据库基本信息
hive (myhive)> desc  database  myhive2;

查看数据库更多详细信息
hive (myhive)> desc database extended  myhive2;
```

- 删除数据库

```
删除一个空数据库，如果数据库下面有数据表，那么就会报错
drop  database  myhive2;
```

```
强制删除数据库，包含数据库下面的表一起删除
drop  database  myhive  cascade;
```

对数据表的操作

对管理表(内部表)的操作：

- 建内部表：

```
hive (myhive)> use myhive; -- 使用myhive数据库
hive (myhive)> create table stu(id int,name string);
hive (myhive)> insert into stu values (1,"zhangsan");
hive (myhive)> insert into stu values (1,"zhangsan"),(2,"lisi");  -- 一次插入多条数据
hive (myhive)> select * from stu;
```

- hive 建表时候的字段类型：

分类	类型	描述	字面量示例
原始类型	BOOLEAN	true/false	TRUE
	TINYINT	1 字节的有符号整数 -128~127	1Y
	SMALLINT	2 个字节的有符号整数，-32768~32767	1S
	INT	4 个字节的带符号整数	1
	BIGINT	8 字节带符号整数	1L
	FLOAT	4 字节单精度浮点数 1.0	

分类	类型	描述	字面量示例
	DOUBLE	8 字节双精度浮点数	1.0
	DEICIMAL	任意精度的带符号小数	1.0
	STRING	字符串，变长	“a”,’ b’
	VARCHAR	变长字符串	“a”,’ b’
	CHAR	固定长度字符串	“a”,’ b’
	BINARY	字节数组	无法表示
	TIMESTAMP	时间戳，毫秒值精度	122327493795
	DATE	日期	‘2016-03-29’
	INTERVAL	时间频率间隔	
复杂类型	ARRAY	有序的的同类型的集合	array(1,2)
	MAP	key-value,key 必须为原始类型，value 可以任意类型	map(‘a’ ,1,’ b’ ,2)
	STRUCT	字段集合,类型可以不同	struct(‘1’ ,1,1.0), named_stract(‘col1’ ,’ 1’ ,’ col2’ ,1,’ clo3’ ,1.0)
	UNION	在有限取值范围内的一个值	create_union(1,’ a’ ,63)

对 decimal 类型简单解释下：

用法：decimal(11,2) 代表最多有 11 位数字，其中后 2 位是小数，整数部分是 9 位；如果整数部分超过 9 位，则这个字段就会变成 null；如果小数部分不足 2 位，

则后面用 0 补齐两位，如果小数部分超过两位，则超出部分四舍五入
也可直接写 decimal，后面不指定位数，默认是 decimal(10,0) 整数 10 位，没有小数

- 创建表并指定字段之间的分隔符

```
create  table if not exists stu2(id int ,name string) row format delimited fields terminated by '\t' stored as textfile location '/user/stu2';
```

row format delimited fields terminated by ’\t’ 指定字段分隔符，默认分隔符为 ’\001’
stored as 指定存储格式
location 指定存储位置

- 根据查询结果创建表

```
create table stu3 as select * from stu2;
```

- 根据已经存在的表结构创建表

```
create table stu4 like stu2;
```

- 查询表的结构

```
只查询表内字段及属性
desc stu2;

详细查询
desc formatted  stu2;
```

- 查询创建表的语句

```
show create table stu2;
```

对外部表操作

外部表因为是指定其他的 hdfs 路径的数据加载到表当中来，所以 hive 表会认为自己不完全独占这份数据，所以删除 hive 表的时候，数据仍然存放在 hdfs 当中，不会删掉，只会删除表的元数据

- 构建外部表

```
create external table student (s_id string,s_name string) row format delimited fiel
ds terminated by '\t';
```

- 从本地文件系统向表中加载数据

```
追加操作
load data local inpath '/export/servers/hivedatas/student.csv' into table student;

覆盖操作
load data local inpath '/export/servers/hivedatas/student.csv' overwrite  into tabl
e student;
```

- 从 hdfs 文件系统向表中加载数据

```
load data inpath '/hivedatas/techer.csv' into table techer;

加载数据到指定分区
load data inpath '/hivedatas/techer.csv' into table techer partition(cur_date=20201
210);
```

- 注意：

  1. 使用 load data local 表示从本地文件系统加载，文件会拷贝到 hdfs 上
  2. 使用 load data 表示从 hdfs 文件系统加载，文件会直接移动到 hive 相关目录下，注意不是拷贝过去，因为 hive 认为 hdfs 文件已经有 3 副本了，没必要再次拷贝了
  3. 如果表是分区表，load 时不指定分区会报错
  4. 如果加载相同文件名的文件，会被自动重命名

对分区表的操作

- 创建分区表的语法

```
create table score(s_id string, s_score int) partitioned by (month string);
```

- 创建一个表带多个分区

```
create table score2 (s_id string, s_score int) partitioned by (year string,month st
ring,day string);
```

注意：
hive 表创建的时候可以用 location 指定一个文件或者文件夹，当指定文件夹时，hive 会加载文件夹下的所有文件，当表中无分区时，这个文件夹下不能再有文件夹，否则报错
当表是分区表时，比如 partitioned by (day string)， 则这个文件夹下的每一个文件夹就是一个分区，且文件夹名为 day=20201123 这种格式，然后使用：msck repair table score; 修复表结构，成功之后即可看到数据已经全部加载到表当中去了

- 加载数据到一个分区的表中

```
load data local inpath '/export/servers/hivedatas/score.csv' into table score partition (month='201806');
```

- 加载数据到一个多分区的表中去

```
load data local inpath '/export/servers/hivedatas/score.csv' into table score2 partition(year='2018',month='06',day='01');
```

- 查看分区

```
show  partitions  score;
```

- 添加一个分区

```
alter table score add partition(month='201805');
```

- 同时添加多个分区

```
 alter table score add partition(month='201804') partition(month = '201803');
```

注意：添加分区之后就可以在 hdfs 文件系统当中看到表下面多了一个文件夹

- 删除分区

```
 alter table score drop partition(month = '201806');
```

对分桶表操作

将数据按照指定的字段进行分成多个桶中去，就是按照分桶字段进行哈希划分到多个文件当中去
分区就是分文件夹，分桶就是分文件

分桶优点：
1. 提高 join 查询效率
2. 提高抽样效率

- 开启 hive 的捅表功能

```
set hive.enforce.bucketing=true;
```

- 设置 reduce 的个数

```
set mapreduce.job.reduces=3;
```

- 创建桶表

```
create table course (c_id string,c_name string) clustered by(c_id) into 3 buckets;
```

桶表的数据加载：由于桶表的数据加载通过 hdfs dfs -put 文件或者通过 load data 均不可以，只能通过 insert overwrite 进行加载
所以把文件加载到桶表中，需要先创建普通表，并通过 insert overwrite 的方式将普通表的数据通过查询的方式加载到桶表当中去

- 通过 insert overwrite 给桶表中加载数据

```
insert overwrite table course select * from course_common cluster by(c_id);  -- 最后指定桶字段
```

修改表和删除表

- 修改表名称

```
alter  table  old_table_name  rename  to  new_table_name;
```

- 增加/修改列信息

```
查询表结构
desc score5;

添加列
alter table score5 add columns (mycol string, mysco string);

更新列
alter table score5 change column mysco mysconew int;
```

- 删除表操作

```
drop table score5;
```

- 清空表操作

```
truncate table score6;
```

```
说明：只能清空管理表，也就是内部表；清空外部表，会产生错误
```

注意：truncate 和 drop：
如果 hdfs 开启了回收站，drop 删除的表数据是可以从回收站恢复的，表结构恢复不了，需要自己重新创建；truncate 清空的表是不进回收站的，所以无法恢复 truncate 清空的表
所以 truncate 一定慎用，一旦清空将无力回天

向 hive 表中加载数据

- 直接向分区表中插入数据

```
insert into table score partition(month ='201807') values ('001','002','100');
```

- 通过 load 方式加载数据

```
 load data local inpath '/export/servers/hivedatas/score.csv' overwrite into table score partition(month='201806');
```

- 通过查询方式加载数据

```
insert overwrite table score2 partition(month = '201806') select s_id,c_id,s_score from score1;
```

- 查询语句中创建表并加载数据

```
create table score2 as select * from score1;
```

- 在创建表是通过 location 指定加载数据的路径

```
create external table score6 (s_id string,c_id string,s_score int) row format delimited fields terminated by ',' location '/myscore';
```

- export 导出与 import 导入 hive 表数据（内部表操作）

```
create table techer2 like techer; --依据已有表结构创建表

export table techer to  '/export/techer';

import table techer2 from '/export/techer';
```

hive 表中数据导出

- insert 导出

```
将查询的结果导出到本地
insert overwrite local directory '/export/servers/exporthive' select * from score;

将查询的结果格式化导出到本地
insert overwrite local directory '/export/servers/exporthive' row format delimited
fields terminated by '\t' collection items terminated by '#' select * from student;

将查询的结果导出到 HDFS 上(没有 local)
insert overwrite directory '/export/servers/exporthive' row format delimited fields
 terminated by '\t' collection items terminated by '#' select * from score;
```

- Hadoop 命令导出到本地

```
dfs -get /export/servers/exporthive/000000_0 /export/servers/exporthive/local.txt;
```

- hive shell 命令导出

```
基本语法：（hive -f/-e 执行语句或者脚本 > file）

hive -e "select * from myhive.score;" > /export/servers/exporthive/score.txt

hive -f export.sh > /export/servers/exporthive/score.txt
```

- export 导出到 HDFS 上

```
export table score to '/export/exporthive/score';
```

hive 的 DQL 查询语法

单表查询

```
SELECT [ALL | DISTINCT] select_expr, select_expr, ...
FROM table_reference
[WHERE where_condition]
[GROUP BY col_list [HAVING condition]]
[CLUSTER BY col_list
  | [DISTRIBUTE BY col_list] [SORT BY| ORDER BY col_list]
]
[LIMIT number]
```

注意：
1、order by 会对输入做全局排序，因此只有一个 reducer，会导致当输入规模较大时，需要较长的计算时间。
2、sort by 不是全局排序，其在数据进入 reducer 前完成排序。因此，如果用 sort by 进行排序，并且设置 mapred.reduce.tasks>1，则 sort by 只保证每个 reducer 的输出有序，不保证全局有序。
3、distribute by(字段)根据指定的字段将数据分到不同的 reducer，且分发算法是 hash 散列。
4、Cluster by(字段) 除了具有 Distribute by 的功能外，还会对该字段进行排序。因此，如果分桶和 sort 字段是同一个时，此时，cluster by = distribute by + sort by

- WHERE 语句

```
select * from score where s_score < 60;
```

注意：
小于某个值是不包含 null 的，如上查询结果是把 s_score 为 null 的行剔除的

- GROUP BY 分组

```
select s_id ,avg(s_score) from score group by s_id;
```

```
分组后对数据进行筛选，使用 having
 select s_id ,avg(s_score) avgscore from score group by s_id having avgscore > 85;
```

注意：
如果使用 group by 分组，则 select 后面只能写分组的字段或者聚合函数
where 和 having 区别：
1 having 是在 group by 分完组之后再对数据进行筛选，所以 having 要筛选的字段只能是分组字段或者聚合函数

2 where 是从数据表中的字段直接进行的筛选的，所以不能跟在 gruop by 后面，也不能使用聚合函数

- join 连接

```
INNER JOIN 内连接：只有进行连接的两个表中都存在与连接条件相匹配的数据才会被保留下来
select * from techer t [inner] join course c on t.t_id = c.t_id; -- inner 可省略

LEFT OUTER JOIN 左外连接：左边所有数据会被返回，右边符合条件的被返回
select * from techer t left join course c on t.t_id = c.t_id; -- outer 可省略

RIGHT OUTER JOIN 右外连接：右边所有数据会被返回，左边符合条件的被返回、
select * from techer t right join course c on t.t_id = c.t_id;

FULL OUTER JOIN 满外(全外)连接：将会返回所有表中符合条件的所有记录。如果任一表的指定字段没有符合条件的值的话，那么就使用 NULL 值替代。
SELECT * FROM techer t FULL JOIN course c ON t.t_id = c.t_id ;
```

注：1. hive2 版本已经支持不等值连接，就是 join on 条件后面可以使用大于小于符号了;并且也支持 join on 条件后跟 or（早前版本 on 后只支持 = 和 and，不支持 > < 和 or）
2. 如 hive 执行引擎使用 MapReduce，一个 join 就会启动一个 job，一条 sql 语句中如有多个 join，则会启动多个 job
注意：表之间用逗号(,)连接和 inner join 是一样的

```
select * from table_a,table_b where table_a.id=table_b.id;
```

它们的执行效率没有区别，只是书写方式不同，用逗号是 sql 89 标准，join 是 sql 92 标准。用逗号连接后面过滤条件用 where ，用 join 连接后面过滤条件是 on。

- order by 排序

```
全局排序，只会有一个 reduce
ASC（ascend）: 升序（默认） DESC（descend）: 降序
SELECT * FROM student s LEFT JOIN score sco ON s.s_id = sco.s_id ORDER BY sco.s_score DESC;
```

注意：order by 是全局排序，所以最后只有一个 reduce，也就是在一个节点执行，如果数据量太大，就会耗费较长时间

- sort by 局部排序

```
每个 MapReduce 内部进行排序，对全局结果集来说不是排序。

设置 reduce 个数
```

```
set mapreduce.job.reduces=3;

查看设置 reduce 个数
set mapreduce.job.reduces;

查询成绩按照成绩降序排列
select * from score sort by s_score;

将查询结果导入到文件中（按照成绩降序排列）
insert overwrite local directory '/export/servers/hivedatas/sort' select * from sco
re sort by s_score;
```

- distribute by 分区排序

```
distribute by：类似 MR 中 partition，进行分区，结合 sort by 使用

设置 reduce 的个数，将我们对应的 s_id 划分到对应的 reduce 当中去
set mapreduce.job.reduces=7;

通过 distribute by  进行数据的分区
select * from score distribute by s_id sort by s_score;
```

注意：Hive 要求 distribute by 语句要写在 sort by 语句之前

- cluster by

```
当 distribute by 和 sort by 字段相同时，可以使用 cluster by 方式.
cluster by 除了具有 distribute by 的功能外还兼具 sort by 的功能。但是排序只能是正序排序，不
能指定排序规则为 ASC 或者 DESC。

以下两种写法等价
select * from score cluster by s_id;
select * from score distribute by s_id sort by s_id;
```

Hive 函数

聚合函数

```
hive 支持 count(),max(),min(),sum(),avg() 等常用的聚合函数
```

注意：
聚合操作时要注意 null 值
count(*) 包含 null 值，统计所有行数

```
count(id) 不包含 null 值
min 求最小值是不包含 null，除非所有值都是 null
avg 求平均值也是不包含 null
```

- 非空集合总体变量函数：var_pop

```
语法：var_pop(col)
返回值：double
说明：统计结果集中 col 非空集合的总体变量（忽略 null）
```

- 非空集合样本变量函数：var_samp

```
语法：var_samp (col)
返回值：double
说明：统计结果集中 col 非空集合的样本变量（忽略 null）
```

- 总体标准偏离函数：stddev_pop

```
语法：stddev_pop(col)
返回值：double
说明：该函数计算总体标准偏离，并返回总体变量的平方根，其返回值与 VAR_POP 函数的平方根相同
```

- 中位数函数：percentile

```
语法：percentile(BIGINT col, p)
返回值：double
说明：求准确的第 pth 个百分位数，p 必须介于 0 和 1 之间，但是 col 字段目前只支持整数，不支持浮点数类型
```

关系运算

```
支持：等值(=)、不等值(!= 或 <>)、小于(<)、小于等于(<=)、大于(>)、大于等于(>=)
```

```
空值判断(is null)、非空判断(is not null)
```

- LIKE 比较：LIKE

```
语法：A LIKE B
操作类型：strings
描述：如果字符串 A 或者字符串 B 为 NULL，则返回 NULL；如果字符串 A 符合表达式 B 的正则语法，则为 TRUE；否则为 FALSE。B 中字符”_”表示任意单个字符，而字符”%”表示任意数量的字符。
```

- JAVA 的 LIKE 操作：RLIKE

语法：A RLIKE B
操作类型：strings
描述：如果字符串 A 或者字符串 B 为 NULL，则返回 NULL；如果字符串 A 符合 JAVA 正则表达式 B 的正则语法，则为 TRUE；否则为 FALSE。

- REGEXP 操作：REGEXP

```
语法：A REGEXP B
操作类型：strings
描述：功能与 RLIKE 相同
示例：select 1 from tableName where 'footbar' REGEXP '^f.*r$';
结果：1
```

数学运算

支持所有数值类型：加(+)、减(-)、乘(*)、除(/)、取余(%)、位与(&)、位或(|)、位异或(^)、位取反(~)

逻辑运算

支持：逻辑与(and)、逻辑或(or)、逻辑非(not)

数值运算

- 取整函数：round

```
语法：round(double a)
返回值：BIGINT
说明：返回 double 类型的整数值部分 （遵循四舍五入）
示例：select round(3.1415926) from tableName;
结果：3
```

- 指定精度取整函数：round

```
语法：round(double a, int d)
返回值：DOUBLE
说明：返回指定精度 d 的 double 类型
hive> select round(3.1415926,4) from tableName;
3.1416
```

- 向下取整函数：floor

```
语法: floor(double a)
返回值: BIGINT
说明: 返回等于或者小于该 double 变量的最大的整数
hive> select floor(3.641) from tableName;
3
```

- 向上取整函数：ceil

```
语法: ceil(double a)
返回值: BIGINT
说明: 返回等于或者大于该 double 变量的最小的整数
hive> select ceil(3.1415926) from tableName;
4
```

- 取随机数函数：rand

```
语法: rand(),rand(int seed)
返回值: double
说明: 返回一个 0 到 1 范围内的随机数。如果指定种子 seed，则会等到一个稳定的随机数序列
hive> select rand() from tableName; -- 每次执行此语句得到的结果都不同
0.5577432776034763

hive> select rand(100) ;  -- 只要指定种子，每次执行此语句得到的结果一样的
0.7220096548596434
```

- 自然指数函数：exp

```
语法: exp(double a)
返回值: double
说明: 返回自然对数 e 的 a 次方
hive> select exp(2) ;
7.38905609893065
```

- 以 10 为底对数函数：log10

```
语法: log10(double a)
返回值: double
说明: 返回以 10 为底的 a 的对数
hive> select log10(100) ;
2.0
```

此外还有：以 2 为底对数函数：log2()、对数函数：log()

- 幂运算函数：pow

```
语法: pow(double a, double p)
返回值: double
说明: 返回 a 的 p 次幂
hive> select pow(2,4) ;
16.0
```

- 开平方函数: sqrt

```
语法: sqrt(double a)
返回值: double
说明: 返回 a 的平方根
hive> select sqrt(16) ;
4.0
```

- 二进制函数: bin

```
语法: bin(BIGINT a)
返回值: string
说明: 返回 a 的二进制代码表示
hive> select bin(7) ;
111
```

十六进制函数: hex()、将十六进制转化为字符串函数: unhex()
进制转换函数: conv(bigint num, int from_base, int to_base) 说明: 将数值 num 从 from_base 进制转化到 to_base 进制
此外还有很多数学函数： 绝对值函数: abs()、正取余函数: pmod()、正弦函数: sin()、反正弦函数: asin()、余弦函数: cos()、反余弦函数: acos()、positive 函数: positive()、negative 函数: negative()

条件函数

- If 函数: if

```
语法: if(boolean testCondition, T valueTrue, T valueFalseOrNull)
返回值: T
说明: 当条件 testCondition 为 TRUE 时，返回 valueTrue；否则返回 valueFalseOrNull
hive> select if(1=2,100,200) ;
200
hive> select if(1=1,100,200) ;
100
```

- 非空查找函数: coalesce

```
语法: coalesce(T v1, T v2, …)
返回值: T
说明: 返回参数中的第一个非空值；如果所有值都为 NULL，那么返回 NULL
hive> select coalesce(null,'100','50') ;
100
```

- 条件判断函数：case when（两种写法，其一）

```
语法: case when a then b [when c then d]* [else e] end
返回值: T
说明: 如果 a 为 TRUE,则返回 b；如果 c 为 TRUE，则返回 d；否则返回 e
hive> select case when 1=2 then 'tom' when 2=2 then 'mary' else 'tim' end from tabl
eName;
mary
```

- 条件判断函数：case when（两种写法，其二）

```
语法: case a when b then c [when d then e]* [else f] end
返回值: T
说明: 如果 a 等于 b，那么返回 c；如果 a 等于 d，那么返回 e；否则返回 f
hive> Select case 100 when 50 then 'tom' when 100 then 'mary' else 'tim' end from t
ableName;
mary
```

日期函数

注：以下 SQL 语句中的 from tableName 可去掉，不影响查询结果

- 1. 获取当前 UNIX 时间戳函数: unix_timestamp

```
语法: unix_timestamp()
返回值: bigint
说明: 获得当前时区的 UNIX 时间戳
hive> select unix_timestamp() from tableName;
1616906976
```

- 2. UNIX 时间戳转日期函数: from_unixtime

```
语法: from_unixtime(bigint unixtime[, string format])
返回值: string
说明: 转化 UNIX 时间戳（从 1970-01-01 00:00:00 UTC 到指定时间的秒数）到当前时区的时间格式
hive> select from_unixtime(1616906976,'yyyyMMdd') from tableName;
20210328
```

- 3. 日期转 UNIX 时间戳函数: unix_timestamp

```
语法: unix_timestamp(string date)
返回值: bigint
说明: 转换格式为"yyyy-MM-dd HH:mm:ss"的日期到 UNIX 时间戳。如果转化失败，则返回 0。
hive>  select unix_timestamp('2021-03-08 14:21:15') from tableName;
1615184475
```

- 4. 指定格式日期转 UNIX 时间戳函数: unix_timestamp

```
语法: unix_timestamp(string date, string pattern)
返回值: bigint
说明: 转换 pattern 格式的日期到 UNIX 时间戳。如果转化失败，则返回 0。
hive>  select unix_timestamp('2021-03-08 14:21:15','yyyyMMdd HH:mm:ss') from tableN
ame;
1615184475
```

- 5. 日期时间转日期函数: to_date

```
语法: to_date(string timestamp)
返回值: string
说明: 返回日期时间字段中的日期部分。
hive> select to_date('2021-03-28 14:03:01') from tableName;
2021-03-28
```

- 6. 日期转年函数: year

```
语法: year(string date)
返回值: int
说明: 返回日期中的年。
hive> select year('2021-03-28 10:03:01') from tableName;
2021
hive> select year('2021-03-28') from tableName;
2021
```

- 7. 日期转月函数: month

```
语法: month (string date)
返回值: int
说明: 返回日期中的月份。
```

```
hive> select month('2020-12-28 12:03:01') from tableName;
12
hive> select month('2021-03-08') from tableName;
8
```

- 8. 日期转天函数：day

```
语法: day (string date)
返回值: int
说明: 返回日期中的天。
hive> select day('2020-12-08 10:03:01') from tableName;
8
hive> select day('2020-12-24') from tableName;
24
```

- 9. 日期转小时函数：hour

```
语法: hour (string date)
返回值: int
说明: 返回日期中的小时。
hive> select hour('2020-12-08 10:03:01') from tableName;
10
```

- 10. 日期转分钟函数：minute

```
语法: minute (string date)
返回值: int
说明: 返回日期中的分钟。
hive> select minute('2020-12-08 10:03:01') from tableName;
3
```

- 11. 日期转秒函数：second

```
语法: second (string date)
返回值: int
说明: 返回日期中的秒。
hive> select second('2020-12-08 10:03:01') from tableName;
1
```

- 12. 日期转周函数：weekofyear

```
语法: weekofyear (string date)
返回值: int
说明: 返回日期在当前的周数。
hive> select weekofyear('2020-12-08 10:03:01') from tableName;
49
```

- 13. 日期比较函数: datediff

```
语法: datediff(string enddate, string startdate)
返回值: int
说明: 返回结束日期减去开始日期的天数。
hive> select datediff('2020-12-08','2020-05-09') from tableName;
213
```

- 14. 日期增加函数: date_add

```
语法: date_add(string startdate, int days)
返回值: string
说明: 返回开始日期 startdate 增加 days 天后的日期。
hive> select date_add('2020-12-08',10) from tableName;
2020-12-18
```

- 15. 日期减少函数: date_sub

```
语法: date_sub (string startdate, int days)
返回值: string
说明: 返回开始日期 startdate 减少 days 天后的日期。
hive> select date_sub('2020-12-08',10) from tableName;
2020-11-28
```

字符串函数

- 1. 字符串长度函数：length

```
语法: length(string A)
返回值: int
说明: 返回字符串 A 的长度
hive> select length('abcedfg') from tableName;
7
```

- 2. 字符串反转函数：reverse

```
语法: reverse(string A)
返回值: string
说明：返回字符串 A 的反转结果
hive> select reverse('abcedfg') from tableName;
gfdecba
```

- 3. 字符串连接函数：concat

```
语法: concat(string A, string B…)
返回值: string
说明：返回输入字符串连接后的结果，支持任意个输入字符串
hive> select concat('abc','def','gh')from tableName;
abcdefgh
```

- 4. 带分隔符字符串连接函数：concat_ws

```
语法: concat_ws(string SEP, string A, string B…)
返回值: string
说明：返回输入字符串连接后的结果，SEP 表示各个字符串间的分隔符
hive> select concat_ws(',','abc','def','gh')from tableName;
abc,def,gh
```

- 5. 字符串截取函数：substr,substring

```
语法: substr(string A, int start),substring(string A, int start)
返回值: string
说明：返回字符串 A 从 start 位置到结尾的字符串
hive> select substr('abcde',3) from tableName;
cde
hive> select substring('abcde',3) from tableName;
cde
hive> select substr('abcde',-1) from tableName; （和 ORACLE 相同）
e
```

- 6. 字符串截取函数：substr,substring

```
语法: substr(string A, int start, int len),substring(string A, int start, int len)
返回值: string
```

```
说明：返回字符串 A 从 start 位置开始，长度为 len 的字符串
hive> select substr('abcde',3,2) from tableName;
cd
hive> select substring('abcde',3,2) from tableName;
cd
hive>select substring('abcde',-2,2) from tableName;
de
```

- 7. 字符串转大写函数：upper,ucase

```
语法: upper(string A) ucase(string A)
返回值: string
说明：返回字符串 A 的大写格式
hive> select upper('abSEd') from tableName;
ABSED
hive> select ucase('abSEd') from tableName;
ABSED
```

- 8. 字符串转小写函数：lower,lcase

```
语法: lower(string A) lcase(string A)
返回值: string
说明：返回字符串 A 的小写格式
hive> select lower('abSEd') from tableName;
absed
hive> select lcase('abSEd') from tableName;
absed
```

- 9. 去空格函数：trim

```
语法: trim(string A)
返回值: string
说明：去除字符串两边的空格
hive> select trim(' abc ') from tableName;
abc
```

- 10. 左边去空格函数：ltrim

```
语法: ltrim(string A)
返回值: string
说明：去除字符串左边的空格
```

```
hive> select ltrim(' abc ') from tableName;
abc
```

- 11.右边去空格函数：rtrim

```
语法: rtrim(string A)
返回值: string
说明: 去除字符串右边的空格
hive> select rtrim(' abc ') from tableName;
abc
```

- 12.正则表达式替换函数：regexp_replace

```
语法: regexp_replace(string A, string B, string C)
返回值: string
说明: 将字符串 A 中的符合 java 正则表达式 B 的部分替换为 C。注意，在有些情况下要使用转义字符,类
似 oracle 中的 regexp_replace 函数。
hive> select regexp_replace('foobar', 'oo|ar', '') from tableName;
fb
```

- 13.正则表达式解析函数：regexp_extract

```
语法: regexp_extract(string subject, string pattern, int index)
返回值: string
说明: 将字符串 subject 按照 pattern 正则表达式的规则拆分，返回 index 指定的字符。
hive> select regexp_extract('foothebar', 'foo(.*?)(bar)', 1) from tableName;
the
hive> select regexp_extract('foothebar', 'foo(.*?)(bar)', 2) from tableName;
bar
hive> select regexp_extract('foothebar', 'foo(.*?)(bar)', 0) from tableName;
foothebar
strong>注意，在有些情况下要使用转义字符，下面的等号要用双竖线转义，这是 java 正则表达式的规则。
select data_field,
regexp_extract(data_field,'.*?bgStart\\=([^&]+)',1) as aaa,
regexp_extract(data_field,'.*?contentLoaded_headStart\\=([^&]+)',1) as bbb,
regexp_extract(data_field,'.*?AppLoad2Req\\=([^&]+)',1) as ccc
from pt_nginx_loginlog_st
where pt = '2021-03-28' limit 2;
```

- 14.URL 解析函数：parse_url

```
语法: parse_url(string urlString, string partToExtract [, string keyToExtract])
返回值: string
说明：返回 URL 中指定的部分。partToExtract 的有效值为：
HOST, PATH, QUERY, REF, PROTOCOL, AUTHORITY, FILE, and USERINFO.
hive> select parse_url
('https://www.tableName.com/path1/p.php?k1=v1&k2=v2#Ref1', 'HOST')
from tableName;
www.tableName.com
hive> select parse_url
('https://www.tableName.com/path1/p.php?k1=v1&k2=v2#Ref1', 'QUERY', 'k1')
 from tableName;
v1
```

- 15. json 解析函数：get_json_object

```
语法: get_json_object(string json_string, string path)
返回值: string
说明：解析 json 的字符串 json_string,返回 path 指定的内容。如果输入的 json 字符串无效，那么返回 NULL。
hive> select  get_json_object('{"store":{"fruit":\[{"weight":8,"type":"apple"},{"weight":9,"type":"pear"}], "bicycle":{"price":19.95,"color":"red"} },"email":"amy@only_for_json_udf_test.net","owner":"amy"}','$.owner') from tableName;
```

- 16. 空格字符串函数：space

```
语法: space(int n)
返回值: string
说明：返回长度为 n 的字符串
hive> select space(10) from tableName;
hive> select length(space(10)) from tableName;
10
```

- 17. 重复字符串函数：repeat

```
语法: repeat(string str, int n)
返回值: string
说明：返回重复 n 次后的 str 字符串
hive> select repeat('abc',5) from tableName;
abcabcabcabcabc
```

- 18. 首字符 ascii 函数：ascii

```
语法: ascii(string str)
返回值: int
说明: 返回字符串 str 第一个字符的 ascii 码
hive> select ascii('abcde') from tableName;
97
```

- 19. 左补足函数：lpad

```
语法: lpad(string str, int len, string pad)
返回值: string
说明: 将 str 进行用 pad 进行左补足到 len 位
hive> select lpad('abc',10,'td') from tableName;
tdtdtdtabc
注意：与 GP，ORACLE 不同，pad 不能默认
```

- 20. 右补足函数：rpad

```
语法: rpad(string str, int len, string pad)
返回值: string
说明: 将 str 进行用 pad 进行右补足到 len 位
hive> select rpad('abc',10,'td') from tableName;
abctdtdtdt
```

- 21. 分割字符串函数: split

```
语法: split(string str, string pat)
返回值: array
说明: 按照 pat 字符串分割 str，会返回分割后的字符串数组
hive> select split('abtcdtef','t') from tableName;
["ab","cd","ef"]
```

- 22. 集合查找函数: find_in_set

```
语法: find_in_set(string str, string strList)
返回值: int
说明: 返回 str 在 strlist 第一次出现的位置，strlist 是用逗号分割的字符串。如果没有找该 str 字符，则返回 0
hive> select find_in_set('ab','ef,ab,de') from tableName;
2
hive> select find_in_set('at','ef,ab,de') from tableName;
0
```

复合类型构建操作

- Map 类型构建：map

```
语法：map (key1, value1, key2, value2, …)
说明：根据输入的 key 和 value 对构建 map 类型
hive> Create table mapTable as select map('100','tom','200','mary') as t from table
Name;
hive> describe mapTable;
t       map<string ,string>
hive> select t from tableName;
{"100":"tom","200":"mary"}
```

- 2. Struct 类型构建：struct

```
语法：struct(val1, val2, val3, …)
说明：根据输入的参数构建结构体 struct 类型
hive> create table struct_table as select struct('tom','mary','tim') as t from tabl
eName;
hive> describe struct_table;
t       struct<col1:string ,col2:string,col3:string>
hive> select t from tableName;
{"col1":"tom","col2":"mary","col3":"tim"}
```

- 3. array 类型构建：array

```
语法：array(val1, val2, …)
说明：根据输入的参数构建数组 array 类型
hive> create table arr_table as select array("tom","mary","tim") as t from tableNam
e;
hive> describe tableName;
t       array<string>
hive> select t from tableName;
["tom","mary","tim"]
```

复杂类型访问操作

- 1. array 类型访问：A[n]

```
语法: A[n]
操作类型: A 为 array 类型，n 为 int 类型
说明：返回数组 A 中的第 n 个变量值。数组的起始下标为 0。比如，A 是个值为['foo', 'bar']的数组类型，那么 A[0]将返回'foo',而 A[1]将返回'bar'
hive> create table arr_table2 as select array("tom","mary","tim") as t
 from tableName;
hive> select t[0],t[1] from arr_table2;
tom     mary    tim
```

- 2. map 类型访问：M[key]

```
语法: M[key]
操作类型: M 为 map 类型，key 为 map 中的 key 值
说明：返回 map 类型 M 中，key 值为指定值的 value 值。比如，M 是值为{'f' -> 'foo', 'b' -> 'bar', 'all' -> 'foobar'}的 map 类型，那么 M['all']将会返回'foobar'
hive> Create table map_table2 as select map('100','tom','200','mary') as t from tableName;
hive> select t['200'],t['100'] from map_table2;
mary    tom
```

- 3. struct 类型访问：S.x

```
语法: S.x
操作类型: S 为 struct 类型
说明：返回结构体 S 中的 x 字段。比如，对于结构体 struct foobar {int foo, int bar}，foobar.foo 返回结构体中的 foo 字段
hive> create table str_table2 as select struct('tom','mary','tim') as t from tableName;
hive> describe tableName;
t       struct<col1:string ,col2:string,col3:string>
hive> select t.col1,t.col3 from str_table2;
tom     tim
```

复杂类型长度统计函数

- 1. Map 类型长度函数：size(Map<k .V>)

```
语法: size(Map<k .V>)
返回值: int
说明: 返回 map 类型的长度
```

```
hive> select size(t) from map_table2;
2
```

- 2. array 类型长度函数：size(Array)

```
语法：size(Array<T>)
返回值：int
说明：返回 array 类型的长度
hive> select size(t) from arr_table2;
4
```

- 3. 类型转换函数 ***

```
类型转换函数：cast
语法：cast(expr as <type>)
返回值：Expected "=" to follow "type"
说明：返回转换后的数据类型
hive> select cast('1' as bigint) from tableName;
1
```

hive 当中的 lateral view 与 explode 以及 reflect 和窗口函数

使用 explode 函数将 hive 表中的 Map 和 Array 字段数据进行拆分

lateral view 用于和 split、explode 等 UDTF 一起使用的，能将一行数据拆分成多行数据，在此基础上可以对拆分的数据进行聚合，lateral view 首先为原始表的每行调用 UDTF，UDTF 会把一行拆分成一行或者多行，lateral view 在把结果组合，产生一个支持别名表的虚拟表。

其中 explode 还可以用于将 hive 一列中复杂的 array 或者 map 结构拆分成多行

需求：现在有数据格式如下

```
zhangsan child1,child2,child3,child4 k1:v1,k2:v2

lisi child5,child6,child7,child8 k3:v3,k4:v4
```

字段之间使用\t 分割，需求将所有的 child 进行拆开成为一列

```
+----------+--+
| mychild  |
+----------+--+
```

```
| child1    |
| child2    |
| child3    |
| child4    |
| child5    |
| child6    |
| child7    |
| child8    |
+-----------+--+
```

将 map 的 key 和 value 也进行拆开，成为如下结果

```
+-----------+-------------+--+
| mymapkey  | mymapvalue  |
+-----------+-------------+--+
| k1        | v1          |
| k2        | v2          |
| k3        | v3          |
| k4        | v4          |
+-----------+-------------+--+
```

- 1. 创建 hive 数据库

```
创建 hive 数据库
hive (default)> create database hive_explode;
hive (default)> use hive_explode;
```

- 2. 创建 hive 表，然后使用 explode 拆分 map 和 array

```
hive (hive_explode)> create  table t3(name string,children array<string>,address Ma
p<string,string>) row format delimited fields terminated by '\t'  collection items
terminated by ',' map keys terminated by ':' stored as textFile;
```

- 3. 加载数据

```
node03 执行以下命令创建表数据文件
 mkdir -p /export/servers/hivedatas/
 cd /export/servers/hivedatas/
 vim maparray
内容如下:
zhangsan child1,child2,child3,child4 k1:v1,k2:v2
lisi child5,child6,child7,child8 k3:v3,k4:v4
```

```
hive 表当中加载数据
hive (hive_explode)> load data local inpath '/export/servers/hivedatas/maparray' into table t3;
```

- 4. 使用 explode 将 hive 当中数据拆开

```
将 array 当中的数据拆分开
hive (hive_explode)> SELECT explode(children) AS myChild FROM t3;

将 map 当中的数据拆分开

hive (hive_explode)> SELECT explode(address) AS (myMapKey, myMapValue) FROM t3;
```

使用 explode 拆分 json 字符串

需求：需求：现在有一些数据格式如下：

```
a:shandong,b:beijing,c:hebei|1,2,3,4,5,6,7,8,9|[{"source":"7fresh","monthSales":4900,"userCount":1900,"score":"9.9"},{"source":"jd","monthSales":2090,"userCount":78981,"score":"9.8"},{"source":"jdmart","monthSales":6987,"userCount":1600,"score":"9.0"}]
```

其中字段与字段之间的分隔符是 |

我们要解析得到所有的 monthSales 对应的值为以下这一列（行转列）

4900

2090

6987

- 1. 创建 hive 表

```
hive (hive_explode)> create table explode_lateral_view
                   > (`area` string,
                   > `goods_id` string,
                   > `sale_info` string)
                   > ROW FORMAT DELIMITED
                   > FIELDS TERMINATED BY '|'
                   > STORED AS textfile;
```

- 2. 准备数据并加载数据

准备数据如下

```
cd /export/servers/hivedatas
vim explode_json
```

```
a:shandong,b:beijing,c:hebei|1,2,3,4,5,6,7,8,9|[{"source":"7fresh","monthSales":4900,"userCount":1900,"score":"9.9"},{"source":"jd","monthSales":2090,"userCount":78981,"score":"9.8"},{"source":"jdmart","monthSales":6987,"userCount":1600,"score":"9.0"}]
```

加载数据到 hive 表当中去

```
hive (hive_explode)> load data local inpath '/export/servers/hivedatas/explode_json' overwrite into table explode_lateral_view;
```

- 3. 使用 explode 拆分 Array

```
hive (hive_explode)> select explode(split(goods_id,',')) as goods_id from explode_lateral_view;
```

- 4. 使用 explode 拆解 Map

```
hive (hive_explode)> select explode(split(area,',')) as area from explode_lateral_view;
```

- 5. 拆解 json 字段

```
hive (hive_explode)> select explode(split(regexp_replace(regexp_replace(sale_info,'\\[\\{',''),'}]',''),'},\\{')) as  sale_info from explode_lateral_view;
```

然后我们想用 get_json_object 来获取 key 为 monthSales 的数据：

```
hive (hive_explode)> select get_json_object(explode(split(regexp_replace(regexp_replace(sale_info,'\\[\\{',''),'}]',''),'},\\{')),'$.monthSales') as  sale_info from explode_lateral_view;
```

然后挂了

```
FAILED: SemanticException [Error 10081]: UDTF's are not supported outside the SELECT clause, nor nested in expressions
```

UDTF explode 不能写在别的函数内

如果你这么写，想查两个字段，

```
select explode(split(area,',')) as area,good_id from explode_lateral_view;
```

会报错

```
FAILED: SemanticException 1:40 Only a single expression in the SELECT clause is sup
ported with UDTF's. Error encountered near token 'good_id'
使用 UDTF 的时候，只支持一个字段，这时候就需要 LATERAL VIEW 出场了
```

配合 LATERAL VIEW 使用

配合 lateral view 查询多个字段

```
hive (hive_explode)> select goods_id2,sale_info from explode_lateral_view LATERAL V
IEW explode(split(goods_id,','))goods as goods_id2;
```

```
其中 LATERAL VIEW explode(split(goods_id,','))goods 相当于一个虚拟表，与原表
explode_lateral_view 笛卡尔积关联
```

也可以多重使用

```
hive (hive_explode)> select goods_id2,sale_info,area2
                     from explode_lateral_view
                     LATERAL VIEW explode(split(goods_id,','))goods as goods_id2
                     LATERAL VIEW explode(split(area,','))area as area2;也是三个表笛卡
尔积的结果
```

最终，我们可以通过下面的句子，把这个 json 格式的一行数据，完全转换成二维表的方式展现

```
hive (hive_explode)> select get_json_object(concat('{',sale_info_1,'}'),'$.source')
 as source,get_json_object(concat('{',sale_info_1,'}'),'$.monthSales') as monthSale
s,get_json_object(concat('{',sale_info_1,'}'),'$.userCount') as monthSales,get_json
_object(concat('{',sale_info_1,'}'),'$.score') as monthSales from explode_lateral_v
iew LATERAL VIEW explode(split(regexp_replace(regexp_replace(sale_info,'\\[\\{',''),
'}]',''),'},\\{'))sale_info as sale_info_1;
```

总结：

Lateral View 通常和 UDTF 一起出现，为了解决 UDTF 不允许在 select 字段的问题。 Multiple Lateral View 可以实现类似笛卡尔乘积。 Outer 关键字可以把不输出的 UDTF 的空结果，输出成 NULL，防止丢失数据。

行转列

相关参数说明：

CONCAT(string A/col, string B/col…)：返回输入字符串连接后的结果，支持任意个输入字符串；

CONCAT_WS(separator, str1, str2,...)：它是一个特殊形式的 CONCAT()。第一个参数剩余参数间的分隔符。分隔符可以是与剩余参数一样的字符串。如果分隔符是 NULL，返回值也将为 NULL。这个函数会跳过分隔符参数后的任何 NULL 和空字符串。分隔符将被加到被连接的字符串之间;

COLLECT_SET(col)：函数只接受基本数据类型，它的主要作用是将某字段的值进行去重汇总，产生 array 类型字段。

数据准备:

name	constellation	blood_type
孙悟空	白羊座	A
老王	射手座	A
宋宋	白羊座	B
猪八戒	白羊座	A
凤姐	射手座	A

需求：把星座和血型一样的人归类到一起。结果如下：

```
射手座,A            老王|凤姐
白羊座,A            孙悟空|猪八戒
白羊座,B            宋宋
```

实现步骤:

- 1. 创建本地 constellation.txt，导入数据

```
node03 服务器执行以下命令创建文件，注意数据使用\t 进行分割
cd /export/servers/hivedatas
vim constellation.txt

数据如下:
孙悟空 白羊座 A
老王 射手座 A
宋宋 白羊座 B
猪八戒 白羊座 A
凤姐 射手座 A
```

- 2. 创建 hive 表并导入数据

```
创建 hive 表并加载数据
hive (hive_explode)> create table person_info(
                     name string,
                     constellation string,
                     blood_type string)
                     row format delimited fields terminated by "\t";

加载数据
hive (hive_explode)> load data local inpath '/export/servers/hivedatas/constellatio
n.txt' into table person_info;
```

- 3. 按需求查询数据

```
hive (hive_explode)> select
                         t1.base,
                         concat_ws('|', collect_set(t1.name)) name
                     from
                         (select
                             name,
                             concat(constellation, "," , blood_type) base
                         from
                             person_info) t1
                     group by
                         t1.base;
```

列转行

所需函数：

EXPLODE(col)：将 hive 一列中复杂的 array 或者 map 结构拆分成多行。

LATERAL VIEW

用法：LATERAL VIEW udtf(expression) tableAlias AS columnAlias

解释：用于和 split, explode 等 UDTF 一起使用，它能够将一列数据拆成多行数据，在此基础上可以对拆分后的数据进行聚合。

数据准备：

```
cd /export/servers/hivedatas
vim movie.txt
文件内容如下：  数据字段之间使用\t 进行分割
《疑犯追踪》  悬疑,动作,科幻,剧情
《Lie to me》  悬疑,警匪,动作,心理,剧情
《战狼 2》  战争,动作,灾难
```

需求：将电影分类中的数组数据展开。结果如下：

```
《疑犯追踪》 悬疑
《疑犯追踪》 动作
《疑犯追踪》 科幻
《疑犯追踪》 剧情
《Lie to me》 悬疑
《Lie to me》 警匪
《Lie to me》 动作
《Lie to me》 心理
《Lie to me》 剧情
《战狼 2》 战争
《战狼 2》 动作
《战狼 2》 灾难
```

实现步骤：

- 1. 创建 hive 表

```
create table movie_info(
    movie string,
    category array<string>)
row format delimited fields terminated by "\t"
collection items terminated by ",";
```

- 2. 加载数据

```
load data local inpath "/export/servers/hivedatas/movie.txt" into table movie_info;
```

- 3. 按需求查询数据

```
select
    movie,
    category_name
from
    movie_info lateral view explode(category) table_tmp as category_name;
```

reflect 函数

reflect 函数可以支持在 sql 中调用 java 中的自带函数，秒杀一切 udf 函数。

需求 1：使用 java.lang.Math 当中的 Max 求两列中最大值

实现步骤：

- 1. 创建 hive 表

```
create table test_udf(col1 int,col2 int) row format delimited fields terminated by
','
```

- 2. 准备数据并加载数据

```
cd /export/servers/hivedatas
vim test_udf
```

```
文件内容如下：
1,2
4,3
6,4
7,5
5,6
```

- 3. 加载数据

```
hive (hive_explode)> load data local inpath '/export/servers/hivedatas/test_udf' ov
erwrite into table test_udf;
```

- 4. 使用 java.lang.Math 当中的 Max 求两列当中的最大值

```
hive (hive_explode)> select reflect("java.lang.Math","max",col1,col2) from test_udf
;
```

需求 2：文件中不同的记录来执行不同的 java 的内置函数

实现步骤：

- 1. 创建 hive 表

```
hive (hive_explode)> create table test_udf2(class_name string,method_name string,co
l1 int , col2 int) row format delimited fields terminated by ',';
```

- 2. 准备数据

```
cd /export/servers/hivedatas
vim test_udf2
```

```
文件内容如下：
java.lang.Math,min,1,2
java.lang.Math,max,2,3
```

- 3. 加载数据

```
hive (hive_explode)> load data local inpath '/export/servers/hivedatas/test_udf2' o
verwrite into table test_udf2;
```

- 4. 执行查询

```
hive (hive_explode)> select reflect(class_name,method_name,col1,col2) from test_udf
2;
```

需求 3：判断是否为数字

实现方式：

使用 apache commons 中的函数，commons 下的 jar 已经包含在 hadoop 的 classpath 中，所以可以直接使用。

```
select reflect("org.apache.commons.lang.math.NumberUtils","isNumber","123")
```

窗口函数与分析函数

在 sql 中有一类函数叫做聚合函数,例如 sum()、avg()、max()等等,这类函数可以将多行数据按照规则聚集为一行,一般来讲聚集后的行数是要少于聚集前的行数的。但是有时我们想要既显示聚集前的数据,又要显示聚集后的数据,这时我们便引入了窗口函数。窗口函数又叫 OLAP 函数/分析函数，窗口函数兼具分组和排序功能。

窗口函数最重要的关键字是 partition by 和 order by。

具体语法如下：over (partition by xxx order by xxx)

sum、avg、min、max

准备数据

```
建表语句：
create table test_t1(
cookieid string,
```

```
createtime string,   --day
pv int
) row format delimited
fields terminated by ',';

加载数据:
load data local inpath '/root/hivedata/test_t1.dat' into table test_t1;

cookie1,2020-04-10,1
cookie1,2020-04-11,5
cookie1,2020-04-12,7
cookie1,2020-04-13,3
cookie1,2020-04-14,2
cookie1,2020-04-15,4
cookie1,2020-04-16,4

开启智能本地模式
SET hive.exec.mode.local.auto=true;
```

SUM 函数和窗口函数的配合使用：结果和 ORDER BY 相关,默认为升序。

```
select cookieid,createtime,pv,
sum(pv) over(partition by cookieid order by createtime) as pv1
from test_t1;

select cookieid,createtime,pv,
sum(pv) over(partition by cookieid order by createtime rows between unbounded prece
ding and current row) as pv2
from test_t1;

select cookieid,createtime,pv,
sum(pv) over(partition by cookieid) as pv3
from test_t1;

select cookieid,createtime,pv,
sum(pv) over(partition by cookieid order by createtime rows between 3 preceding and
 current row) as pv4
from test_t1;

select cookieid,createtime,pv,
sum(pv) over(partition by cookieid order by createtime rows between 3 preceding and
 1 following) as pv5
from test_t1;

select cookieid,createtime,pv,
```

```
sum(pv) over(partition by cookieid order by createtime rows between current row and
 unbounded following) as pv6
from test_t1;
```

```
pv1: 分组内从起点到当前行的 pv 累积,如,11 号的 pv1=10 号的 pv+11 号的 pv, 12 号=10 号+11 号+12
号
pv2: 同 pv1
pv3: 分组内(cookie1)所有的 pv 累加
pv4: 分组内当前行+往前 3 行，如，11 号=10 号+11 号， 12 号=10 号+11 号+12 号，
                        13 号=10 号+11 号+12 号+13 号， 14 号=11 号+12 号+13 号+14 号
pv5: 分组内当前行+往前 3 行+往后 1 行，如，14 号=11 号+12 号+13 号+14 号+15 号=5+7+3+2+4=21
pv6: 分组内当前行+往后所有行，如，13 号=13 号+14 号+15 号+16 号=3+2+4+4=13，
        14 号=14 号+15 号+16 号=2+4+4=10
```

如果不指定 rows between,默认为从起点到当前行;

如果不指定 order by，则将分组内所有值累加;

关键是理解 rows between 含义,也叫做 window 子句：

preceding：往前

following：往后

current row：当前行

unbounded：起点

unbounded preceding 表示从前面的起点

unbounded following：表示到后面的终点

AVG，MIN，MAX，和 SUM 用法一样。

row_number、rank、dense_rank、ntile

准备数据

```
cookie1,2020-04-10,1
cookie1,2020-04-11,5
cookie1,2020-04-12,7
cookie1,2020-04-13,3
cookie1,2020-04-14,2
cookie1,2020-04-15,4
cookie1,2020-04-16,4
cookie2,2020-04-10,2
cookie2,2020-04-11,3
cookie2,2020-04-12,5
cookie2,2020-04-13,6
cookie2,2020-04-14,3
```

```
cookie2,2020-04-15,9
cookie2,2020-04-16,7

CREATE TABLE test_t2 (
cookieid string,
createtime string,   --day
pv INT
) ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
stored as textfile;

加载数据:
load data local inpath '/root/hivedata/test_t2.dat' into table test_t2;
```

- ROW_NUMBER()使用

  ROW_NUMBER()从 1 开始，按照顺序，生成分组内记录的序列。

```
SELECT
cookieid,
createtime,
pv,
ROW_NUMBER() OVER(PARTITION BY cookieid ORDER BY pv desc) AS rn
FROM test_t2;
```

- RANK 和 DENSE_RANK 使用

  RANK() 生成数据项在分组中的排名，排名相等会在名次中留下空位 。

  DENSE_RANK()生成数据项在分组中的排名，排名相等会在名次中不会留下空位。

```
SELECT
cookieid,
createtime,
pv,
RANK() OVER(PARTITION BY cookieid ORDER BY pv desc) AS rn1,
DENSE_RANK() OVER(PARTITION BY cookieid ORDER BY pv desc) AS rn2,
ROW_NUMBER() OVER(PARTITION BY cookieid ORDER BY pv DESC) AS rn3
FROM test_t2
WHERE cookieid = 'cookie1';
```

- NTILE

  有时会有这样的需求:如果数据排序后分为三部分，业务人员只关心其中的一部分，如何将这中间的三分之一数据拿出来呢?NTILE 函数即可以满足。

ntile 可以看成是：把有序的数据集合平均分配到指定的数量（num）个桶中，将桶号分配给每一行。如果不能平均分配，则优先分配较小编号的桶，并且各个桶中能放的行数最多相差 1。

然后可以根据桶号，选取前或后 n 分之几的数据。数据会完整展示出来，只是给相应的数据打标签；具体要取几分之几的数据，需要再嵌套一层根据标签取出。

```
SELECT
cookieid,
createtime,
pv,
NTILE(2) OVER(PARTITION BY cookieid ORDER BY createtime) AS rn1,
NTILE(3) OVER(PARTITION BY cookieid ORDER BY createtime) AS rn2,
NTILE(4) OVER(ORDER BY createtime) AS rn3
FROM test_t2
ORDER BY cookieid,createtime;
```

其他一些窗口函数

lag, lead, first_value, last_value

- LAG
  LAG(col,n,DEFAULT) 用于统计窗口内往上第 n 行值第一个参数为列名，第二个参数为往上第 n 行（可选，默认为 1），第三个参数为默认值（当往上第 n 行为 NULL 时候，取默认值，如不指定，则为 NULL）

```
SELECT cookieid,
createtime,
url,
ROW_NUMBER() OVER(PARTITION BY cookieid ORDER BY createtime) AS rn,
LAG(createtime,1,'1970-01-01 00:00:00') OVER(PARTITION BY cookieid ORDER BY createt
ime) AS last_1_time,
LAG(createtime,2) OVER(PARTITION BY cookieid ORDER BY createtime) AS last_2_time
FROM test_t4;
```

```
last_1_time: 指定了往上第 1 行的值，default 为'1970-01-01 00:00:00'
                  cookie1 第一行，往上 1 行为 NULL,因此取默认值 1970-01-01 00:00:00
                  cookie1 第三行，往上 1 行值为第二行值，2015-04-10 10:00:02
                  cookie1 第六行，往上 1 行值为第五行值，2015-04-10 10:50:01
last_2_time: 指定了往上第 2 行的值，为指定默认值
```

```
        cookie1 第一行，往上 2 行为 NULL
        cookie1 第二行，往上 2 行为 NULL
        cookie1 第四行，往上 2 行为第二行值，2015-04-10 10:00:02
        cookie1 第七行，往上 2 行为第五行值，2015-04-10 10:50:01
```

- LEAD

与 LAG 相反 LEAD(col,n,DEFAULT) 用于统计窗口内往下第 n 行值 第一个参数为列名，第二个参数为往下第 n 行（可选，默认为 1），第三个参数为默认值（当往下第 n 行为 NULL 时候，取默认值，如不指定，则为 NULL）

```
SELECT cookieid,
createtime,
url,
ROW_NUMBER() OVER(PARTITION BY cookieid ORDER BY createtime) AS rn,
LEAD(createtime,1,'1970-01-01 00:00:00') OVER(PARTITION BY cookieid ORDER BY create
time) AS next_1_time,
LEAD(createtime,2) OVER(PARTITION BY cookieid ORDER BY createtime) AS next_2_time
FROM test_t4;
```

- FIRST_VALUE
  取分组内排序后，截止到当前行，第一个值

```
 SELECT cookieid,
 createtime,
 url,
 ROW_NUMBER() OVER(PARTITION BY cookieid ORDER BY createtime) AS rn,
 FIRST_VALUE(url) OVER(PARTITION BY cookieid ORDER BY createtime) AS first1
 FROM test_t4;
```

- LAST_VALUE

取分组内排序后，截止到当前行，最后一个值

```
SELECT cookieid,
createtime,
url,
ROW_NUMBER() OVER(PARTITION BY cookieid ORDER BY createtime) AS rn,
LAST_VALUE(url) OVER(PARTITION BY cookieid ORDER BY createtime) AS last1
FROM test_t4;
```

如果想要取分组内排序后最后一个值，则需要变通一下：

```
SELECT cookieid,
createtime,
```

```
url,
ROW_NUMBER() OVER(PARTITION BY cookieid ORDER BY createtime) AS rn,
LAST_VALUE(url) OVER(PARTITION BY cookieid ORDER BY createtime) AS last1,
FIRST_VALUE(url) OVER(PARTITION BY cookieid ORDER BY createtime DESC) AS last2
FROM test_t4
ORDER BY cookieid,createtime;
```

特别注意 order by

如果不指定 ORDER BY，则进行排序混乱，会出现错误的结果

```
SELECT cookieid,
createtime,
url,
FIRST_VALUE(url) OVER(PARTITION BY cookieid) AS first2
FROM test_t4;
```

cume_dist,percent_rank

这两个序列分析函数不是很常用，注意： 序列函数不支持 WINDOW 子句

- 数据准备

```
d1,user1,1000
d1,user2,2000
d1,user3,3000
d2,user4,4000
d2,user5,5000
```

```
CREATE EXTERNAL TABLE test_t3 (
dept STRING,
userid string,
sal INT
) ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
stored as textfile;
```

```
加载数据：
load data local inpath '/root/hivedata/test_t3.dat' into table test_t3;
```

- CUME_DIST 和 order byd 的排序顺序有关系
  CUME_DIST 小于等于当前值的行数/分组内总行数 order 默认顺序 正序升序 比如，统计小于等于当前薪水的人数，所占总人数的比例

```
SELECT
dept,
userid,
sal,
CUME_DIST() OVER(ORDER BY sal) AS rn1,
CUME_DIST() OVER(PARTITION BY dept ORDER BY sal) AS rn2
FROM test_t3;

rn1: 没有 partition,所有数据均为 1 组，总行数为 5，
     第一行：小于等于 1000 的行数为 1，因此，1/5=0.2
     第三行：小于等于 3000 的行数为 3，因此，3/5=0.6
rn2: 按照部门分组，dpet=d1 的行数为 3,
     第二行：小于等于 2000 的行数为 2，因此，2/3=0.6666666666666666
```

- PERCENT_RANK

  PERCENT_RANK 分组内当前行的 RANK 值-1/分组内总行数-1

  经调研 该函数显示现实意义不明朗 有待于继续考证

```
SELECT
dept,
userid,
sal,
PERCENT_RANK() OVER(ORDER BY sal) AS rn1,    --分组内
RANK() OVER(ORDER BY sal) AS rn11,           --分组内 RANK 值
SUM(1) OVER(PARTITION BY NULL) AS rn12,      --分组内总行数
PERCENT_RANK() OVER(PARTITION BY dept ORDER BY sal) AS rn2
FROM test_t3;

rn1: rn1 = (rn11-1) / (rn12-1)
     第一行,(1-1)/(5-1)=0/4=0
     第二行,(2-1)/(5-1)=1/4=0.25
     第四行,(4-1)/(5-1)=3/4=0.75
rn2: 按照 dept 分组，
      dept=d1 的总行数为 3
      第一行，(1-1)/(3-1)=0
      第三行，(3-1)/(3-1)=1
```

grouping sets,grouping__id,cube,rollup

这几个分析函数通常用于 OLAP 中，不能累加，而且需要根据不同维度上钻和下钻的指标统计，比如，分小时、天、月的 UV 数。

- 数据准备

```
2020-03,2020-03-10,cookie1
2020-03,2020-03-10,cookie5
2020-03,2020-03-12,cookie7
2020-04,2020-04-12,cookie3
2020-04,2020-04-13,cookie2
2020-04,2020-04-13,cookie4
2020-04,2020-04-16,cookie4
2020-03,2020-03-10,cookie2
2020-03,2020-03-10,cookie3
2020-04,2020-04-12,cookie5
2020-04,2020-04-13,cookie6
2020-04,2020-04-15,cookie3
2020-04,2020-04-15,cookie2
2020-04,2020-04-16,cookie1
```

```
CREATE TABLE test_t5 (
month STRING,
day STRING,
cookieid STRING
) ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
stored as textfile;
```

```
加载数据:
load data local inpath '/root/hivedata/test_t5.dat' into table test_t5;
```

- GROUPING SETS

grouping sets 是一种将多个 group by 逻辑写在一个 sql 语句中的便利写法。等价于将不同维度的 GROUP BY 结果集进行 UNION ALL。

GROUPING__ID，表示结果属于哪一个分组集合。

```
SELECT
month,
day,
COUNT(DISTINCT cookieid) AS uv,
GROUPING__ID
FROM test_t5
GROUP BY month,day
GROUPING SETS (month,day)
ORDER BY GROUPING__ID;
```

```
grouping_id 表示这一组结果属于哪个分组集合，
根据 grouping sets 中的分组条件 month，day，1 是代表 month，2 是代表 day
```

```
等价于
SELECT month,NULL,COUNT(DISTINCT cookieid) AS uv,1 AS GROUPING__ID FROM test_t5 GRO
UP BY month UNION ALL
SELECT NULL as month,day,COUNT(DISTINCT cookieid) AS uv,2 AS GROUPING__ID FROM test
_t5 GROUP BY day;
```

再如：

```
SELECT
month,
day,
COUNT(DISTINCT cookieid) AS uv,
GROUPING__ID
FROM test_t5
GROUP BY month,day
GROUPING SETS (month,day,(month,day))
ORDER BY GROUPING__ID;
```

```
等价于
SELECT month,NULL,COUNT(DISTINCT cookieid) AS uv,1 AS GROUPING__ID FROM test_t5 GRO
UP BY month
UNION ALL
SELECT NULL,day,COUNT(DISTINCT cookieid) AS uv,2 AS GROUPING__ID FROM test_t5 GROUP
 BY day
UNION ALL
SELECT month,day,COUNT(DISTINCT cookieid) AS uv,3 AS GROUPING__ID FROM test_t5 GROU
P BY month,day;
```

- CUBE

根据 GROUP BY 的维度的所有组合进行聚合。

```
SELECT
month,
day,
COUNT(DISTINCT cookieid) AS uv,
GROUPING__ID
FROM test_t5
GROUP BY month,day
WITH CUBE
ORDER BY GROUPING__ID;
```

```
等价于
SELECT NULL,NULL,COUNT(DISTINCT cookieid) AS uv,0 AS GROUPING__ID FROM test_t5
UNION ALL
SELECT month,NULL,COUNT(DISTINCT cookieid) AS uv,1 AS GROUPING__ID FROM test_t5 GRO
UP BY month
UNION ALL
SELECT NULL,day,COUNT(DISTINCT cookieid) AS uv,2 AS GROUPING__ID FROM test_t5 GROUP
 BY day
UNION ALL
SELECT month,day,COUNT(DISTINCT cookieid) AS uv,3 AS GROUPING__ID FROM test_t5 GROU
P BY month,day;
```

- ROLLUP

是 CUBE 的子集，以最左侧的维度为主，从该维度进行层级聚合。

```
比如，以 month 维度进行层级聚合：
SELECT
month,
day,
COUNT(DISTINCT cookieid) AS uv,
GROUPING__ID
FROM test_t5
GROUP BY month,day
WITH ROLLUP
ORDER BY GROUPING__ID;

--把 month 和 day 调换顺序，则以 day 维度进行层级聚合：

SELECT
day,
month,
COUNT(DISTINCT cookieid) AS uv,
GROUPING__ID
FROM test_t5
GROUP BY day,month
WITH ROLLUP
ORDER BY GROUPING__ID;
（这里，根据天和月进行聚合，和根据天聚合结果一样，因为有父子关系，如果是其他维度组合的话，就
会不一样）
```

七、Hive 执行计划

Hive SQL 的执行计划描述 SQL 实际执行的整体轮廓，通过执行计划能了解 SQL 程序在转换成相应计算引擎的执行逻辑，掌握了执行逻辑也就能更好地把握程序出现的瓶颈点，从而能够实现更有针对性的优化。此外还能帮助开发者识别看似等价的 SQL 其实是不等价的，看似不等价的 SQL 其实是等价的 SQL。**可以说执行计划是打开 SQL 优化大门的一把钥匙**。

要想学 SQL 执行计划，就需要学习查看执行计划的命令：`explain`，在查询语句的 SQL 前面加上关键字 explain 是查看执行计划的基本方法。

学会 explain，能够给我们工作中使用 hive 带来极大的便利！

查看 SQL 的执行计划

Hive 提供的执行计划目前可以查看的信息有以下几种：

- `explain`：查看执行计划的基本信息；
- `explain dependency`：dependency 在 explain 语句中使用会产生有关计划中输入的额外信息。它显示了输入的各种属性；
- `explain authorization`：查看 SQL 操作相关权限的信息；
- `explain vectorization`：查看 SQL 的向量化描述信息，显示为什么未对 Map 和 Reduce 进行矢量化。从 Hive 2.3.0 开始支持；
- `explain analyze`：用实际的行数注释计划。从 Hive 2.2.0 开始支持；
- `explain cbo`：输出由 Calcite 优化器生成的计划。CBO 从 Hive 4.0.0 版本开始支持；
- `explain locks`：这对于了解系统将获得哪些锁以运行指定的查询很有用。LOCKS 从 Hive 3.2.0 开始支持；
- `explain ast`：输出查询的抽象语法树。AST 在 Hive 2.1.0 版本删除了，存在 bug，转储 AST 可能会导致 OOM 错误，将在 4.0.0 版本修复；
- `explain extended`：加上 extended 可以输出有关计划的额外信息。这通常是物理信息，例如文件名，这些额外信息对我们用处不大；

1. explain 的用法

Hive 提供了 explain 命令来展示一个查询的执行计划，这个执行计划对于我们了解底层原理，Hive 调优，排查数据倾斜等很有帮助。

使用语法如下：

```
explain query;
```

在 hive cli 中输入以下命令(hive 2.3.7)：

```
explain select sum(id) from test1;
```

得到结果：

```
STAGE DEPENDENCIES:
  Stage-1 is a root stage
  Stage-0 depends on stages: Stage-1

STAGE PLANS:
  Stage: Stage-1
    Map Reduce
      Map Operator Tree:
          TableScan
            alias: test1
            Statistics: Num rows: 6 Data size: 75 Basic stats: COMPLETE Column stat
s: NONE
            Select Operator
              expressions: id (type: int)
              outputColumnNames: id
              Statistics: Num rows: 6 Data size: 75 Basic stats: COMPLETE Column st
ats: NONE
              Group By Operator
                aggregations: sum(id)
                mode: hash
                outputColumnNames: _col0
                Statistics: Num rows: 1 Data size: 8 Basic stats: COMPLETE Column s
tats: NONE
                Reduce Output Operator
                  sort order:
                  Statistics: Num rows: 1 Data size: 8 Basic stats: COMPLETE Column
 stats: NONE
                  value expressions: _col0 (type: bigint)
      Reduce Operator Tree:
        Group By Operator
          aggregations: sum(VALUE._col0)
          mode: mergepartial
          outputColumnNames: _col0
          Statistics: Num rows: 1 Data size: 8 Basic stats: COMPLETE Column stats:
NONE
          File Output Operator
            compressed: false
            Statistics: Num rows: 1 Data size: 8 Basic stats: COMPLETE Column stats
: NONE
```

```
            table:
                input format: org.apache.hadoop.mapred.SequenceFileInputFormat
                output format: org.apache.hadoop.hive.ql.io.HiveSequenceFileOutputF
ormat
                serde: org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe

  Stage: Stage-0
    Fetch Operator
      limit: -1
      Processor Tree:
        ListSink
```

看完以上内容有什么感受，是不是感觉都看不懂，不要着急，下面将会详细讲解每个参数，相信你学完下面的内容之后再看 explain 的查询结果将游刃有余。

一个 HIVE 查询被转换为一个由一个或多个 stage 组成的序列（有向无环图 DAG）。这些 stage 可以是 MapReduce stage，也可以是负责元数据存储的 stage，也可以是负责文件系统的操作（比如移动和重命名）的 stage。

我们将上述结果拆分看，先从最外层开始，包含两个大的部分：

1. stage dependencies： 各个 stage 之间的依赖性
2. stage plan： 各个 stage 的执行计划

先看第一部分 stage dependencies ，包含两个 stage，Stage-1 是根 stage，说明这是开始的 stage，Stage-0 依赖 Stage-1，Stage-1 执行完成后执行 Stage-0。

再看第二部分 stage plan，里面有一个 Map Reduce，一个 MR 的执行计划分为两个部分：

1. Map Operator Tree： MAP 端的执行计划树
2. Reduce Operator Tree： Reduce 端的执行计划树

这两个执行计划树里面包含这条 sql 语句的 operator：

1. TableScan：表扫描操作，map 端第一个操作肯定是加载表，所以就是表扫描操作，常见的属性：
    - alias： 表名称
    - Statistics： 表统计信息，包含表中数据条数，数据大小等
2. Select Operator： 选取操作，常见的属性 ：
    - expressions：需要的字段名称及字段类型
    - outputColumnNames：输出的列名称
    - Statistics：表统计信息，包含表中数据条数，数据大小等

3. Group By Operator：分组聚合操作，常见的属性：
    - aggregations：显示聚合函数信息
    - mode：聚合模式，值有 hash：随机聚合，就是 hash partition；partial：局部聚合；final：最终聚合
    - keys：分组的字段，如果没有分组，则没有此字段
    - outputColumnNames：聚合之后输出列名
    - Statistics： 表统计信息，包含分组聚合之后的数据条数，数据大小等
4. Reduce Output Operator：输出到 reduce 操作，常见属性：
    - sort order：值为空 不排序；值为 + 正序排序，值为 - 倒序排序；值为 +- 排序的列为两列，第一列为正序，第二列为倒序
5. Filter Operator：过滤操作，常见的属性：
    - predicate：过滤条件，如 sql 语句中的 where id>=1，则此处显示(id >= 1)
6. Map Join Operator：join 操作，常见的属性：
    - condition map：join 方式 ，如 Inner Join 0 to 1 Left Outer Join0 to 2
    - keys: join 的条件字段
    - outputColumnNames： join 完成之后输出的字段
    - Statistics： join 完成之后生成的数据条数，大小等
7. File Output Operator：文件输出操作，常见的属性
    - compressed：是否压缩
    - table：表的信息，包含输入输出文件格式化方式，序列化方式等
8. Fetch Operator 客户端获取数据操作，常见的属性：
    - limit，值为 -1 表示不限制条数，其他值为限制的条数

2. explain 的使用场景

本节介绍 explain 能够为我们在生产实践中带来哪些便利及解决我们哪些迷惑

案例一：join 语句会过滤 null 的值吗？

现在，我们在 hive cli 输入以下查询计划语句

```
select a.id,b.user_name from test1 a join test2 b on a.id=b.id;
```

问：上面这条 join 语句会过滤 id 为 null 的值吗

执行下面语句：

```
explain select a.id,b.user_name from test1 a join test2 b on a.id=b.id;
```

我们来看结果（为了适应页面展示，仅截取了部分输出信息）：

```
TableScan
 alias: a
 Statistics: Num rows: 6 Data size: 75 Basic stats: COMPLETE Column stats: NONE
 Filter Operator
    predicate: id is not null (type: boolean)
    Statistics: Num rows: 6 Data size: 75 Basic stats: COMPLETE Column stats: NONE
    Select Operator
        expressions: id (type: int)
        outputColumnNames: _col0
        Statistics: Num rows: 6 Data size: 75 Basic stats: COMPLETE Column stats: N
ONE
        HashTable Sink Operator
           keys:
             0 _col0 (type: int)
             1 _col0 (type: int)
...
```

从上述结果可以看到 predicate: id is not null 这样一行，**说明 join 时会自动过滤掉关联字段为 null 值的情况，但 left join 或 full join 是不会自动过滤 null 值的**，大家可以自行尝试下。

案例二：group by 分组语句会进行排序吗？

看下面这条 sql

```
select id,max(user_name) from test1 group by id;
```

问：**group by 分组语句会进行排序吗**

直接来看 explain 之后结果（为了适应页面展示，仅截取了部分输出信息）

```
 TableScan
    alias: test1
    Statistics: Num rows: 9 Data size: 108 Basic stats: COMPLETE Column stats: NONE
    Select Operator
        expressions: id (type: int), user_name (type: string)
        outputColumnNames: id, user_name
        Statistics: Num rows: 9 Data size: 108 Basic stats: COMPLETE Column stats:
NONE
        Group By Operator
           aggregations: max(user_name)
           keys: id (type: int)
           mode: hash
           outputColumnNames: _col0, _col1
```

```
          Statistics: Num rows: 9 Data size: 108 Basic stats: COMPLETE Column stat
s: NONE
          Reduce Output Operator
            key expressions: _col0 (type: int)
            sort order: +
            Map-reduce partition columns: _col0 (type: int)
            Statistics: Num rows: 9 Data size: 108 Basic stats: COMPLETE Column st
ats: NONE
            value expressions: _col1 (type: string)
 ...
```

我们看 Group By Operator，里面有 keys: id (type: int) 说明按照 id 进行分组的，再往下看还有 sort order: + ，**说明是按照 id 字段进行正序排序的**。

案例三：哪条 sql 执行效率高呢？

观察两条 sql 语句

```
SELECT
 a.id,
 b.user_name
FROM
 test1 a
JOIN test2 b ON a.id = b.id
WHERE
 a.id > 2;

SELECT
 a.id,
 b.user_name
FROM
 (SELECT * FROM test1 WHERE id > 2) a
JOIN test2 b ON a.id = b.id;
```

这两条 sql 语句输出的结果是一样的，但是哪条 sql 执行效率高呢？

有人说第一条 sql 执行效率高，因为第二条 sql 有子查询，子查询会影响性能；

有人说第二条 sql 执行效率高，因为先过滤之后，在进行 join 时的条数减少了，所以执行效率就高了。

到底哪条 sql 效率高呢，我们直接在 sql 语句前面加上 explain，看下执行计划不就知道了嘛！

在第一条 sql 语句前加上 explain，得到如下结果

```
hive (default)> explain select a.id,b.user_name from test1 a join test2 b on a.id=b.
id where a.id >2;
OK
Explain
STAGE DEPENDENCIES:
  Stage-4 is a root stage
  Stage-3 depends on stages: Stage-4
  Stage-0 depends on stages: Stage-3

STAGE PLANS:
  Stage: Stage-4
    Map Reduce Local Work
      Alias -> Map Local Tables:
        $hdt$_0:a
          Fetch Operator
            limit: -1
      Alias -> Map Local Operator Tree:
        $hdt$_0:a
          TableScan
            alias: a
            Statistics: Num rows: 6 Data size: 75 Basic stats: COMPLETE Column stat
s: NONE
            Filter Operator
              predicate: (id > 2) (type: boolean)
              Statistics: Num rows: 2 Data size: 25 Basic stats: COMPLETE Column st
ats: NONE
              Select Operator
                expressions: id (type: int)
                outputColumnNames: _col0
                Statistics: Num rows: 2 Data size: 25 Basic stats: COMPLETE Column
stats: NONE
                HashTable Sink Operator
                  keys:
                    0 _col0 (type: int)
                    1 _col0 (type: int)

  Stage: Stage-3
    Map Reduce
      Map Operator Tree:
          TableScan
            alias: b
            Statistics: Num rows: 6 Data size: 75 Basic stats: COMPLETE Column stat
s: NONE
            Filter Operator
```

```
              predicate: (id > 2) (type: boolean)
              Statistics: Num rows: 2 Data size: 25 Basic stats: COMPLETE Column st
ats: NONE
              Select Operator
                expressions: id (type: int), user_name (type: string)
                outputColumnNames: _col0, _col1
                Statistics: Num rows: 2 Data size: 25 Basic stats: COMPLETE Column
stats: NONE
                Map Join Operator
                  condition map:
                       Inner Join 0 to 1
                  keys:
                    0 _col0 (type: int)
                    1 _col0 (type: int)
                  outputColumnNames: _col0, _col2
                  Statistics: Num rows: 2 Data size: 27 Basic stats: COMPLETE Colum
n stats: NONE
                  Select Operator
                    expressions: _col0 (type: int), _col2 (type: string)
                    outputColumnNames: _col0, _col1
                    Statistics: Num rows: 2 Data size: 27 Basic stats: COMPLETE Col
umn stats: NONE
                    File Output Operator
                      compressed: false
                      Statistics: Num rows: 2 Data size: 27 Basic stats: COMPLETE C
olumn stats: NONE
                      table:
                          input format: org.apache.hadoop.mapred.SequenceFileInputF
ormat
                          output format: org.apache.hadoop.hive.ql.io.HiveSequenceF
ileOutputFormat
                          serde: org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe
      Local Work:
        Map Reduce Local Work

  Stage: Stage-0
    Fetch Operator
      limit: -1
      Processor Tree:
        ListSink
```

在第二条 sql 语句前加上 explain，得到如下结果

```
hive (default)> explain select a.id,b.user_name from(select * from  test1 where id>
2 ) a join test2 b on a.id=b.id;
```

```
OK
Explain
STAGE DEPENDENCIES:
  Stage-4 is a root stage
  Stage-3 depends on stages: Stage-4
  Stage-0 depends on stages: Stage-3

STAGE PLANS:
  Stage: Stage-4
    Map Reduce Local Work
      Alias -> Map Local Tables:
        $hdt$_0:test1
          Fetch Operator
            limit: -1
      Alias -> Map Local Operator Tree:
        $hdt$_0:test1
          TableScan
            alias: test1
            Statistics: Num rows: 6 Data size: 75 Basic stats: COMPLETE Column stat
s: NONE
            Filter Operator
              predicate: (id > 2) (type: boolean)
              Statistics: Num rows: 2 Data size: 25 Basic stats: COMPLETE Column st
ats: NONE
              Select Operator
                expressions: id (type: int)
                outputColumnNames: _col0
                Statistics: Num rows: 2 Data size: 25 Basic stats: COMPLETE Column
stats: NONE
                HashTable Sink Operator
                  keys:
                    0 _col0 (type: int)
                    1 _col0 (type: int)

  Stage: Stage-3
    Map Reduce
      Map Operator Tree:
          TableScan
            alias: b
            Statistics: Num rows: 6 Data size: 75 Basic stats: COMPLETE Column stat
s: NONE
            Filter Operator
              predicate: (id > 2) (type: boolean)
              Statistics: Num rows: 2 Data size: 25 Basic stats: COMPLETE Column st
```

```
ats: NONE
              Select Operator
                expressions: id (type: int), user_name (type: string)
                outputColumnNames: _col0, _col1
                Statistics: Num rows: 2 Data size: 25 Basic stats: COMPLETE Column
stats: NONE
                Map Join Operator
                  condition map:
                       Inner Join 0 to 1
                  keys:
                    0 _col0 (type: int)
                    1 _col0 (type: int)
                  outputColumnNames: _col0, _col2
                  Statistics: Num rows: 2 Data size: 27 Basic stats: COMPLETE Colum
n stats: NONE
                  Select Operator
                    expressions: _col0 (type: int), _col2 (type: string)
                    outputColumnNames: _col0, _col1
                    Statistics: Num rows: 2 Data size: 27 Basic stats: COMPLETE Col
umn stats: NONE
                    File Output Operator
                      compressed: false
                      Statistics: Num rows: 2 Data size: 27 Basic stats: COMPLETE C
olumn stats: NONE
                      table:
                          input format: org.apache.hadoop.mapred.SequenceFileInputF
ormat
                          output format: org.apache.hadoop.hive.ql.io.HiveSequenceF
ileOutputFormat
                          serde: org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe
      Local Work:
        Map Reduce Local Work

  Stage: Stage-0
    Fetch Operator
      limit: -1
      Processor Tree:
        ListSink
```

大家有什么发现，除了表别名不一样，其他的执行计划完全一样，都是先进行 where 条件过滤，在进行 join 条件关联。**说明 hive 底层会自动帮我们进行优化，所以这两条 sql 语句执行效率是一样的**。

以上仅列举了 3 个我们生产中既熟悉又有点迷糊的例子，explain 还有很多其他的用途，如查看 stage 的依赖情况、排查数据倾斜、hive 调优等，小伙伴们可以自行尝试。

2. explain dependency 的用法

explain dependency 用于描述一段 SQL 需要的数据来源，输出是一个 json 格式的数据，里面包含以下两个部分的内容：

- input_partitions：描述一段 SQL 依赖的数据来源表分区，里面存储的是分区名的列表，如果整段 SQL 包含的所有表都是非分区表，则显示为空。
- input_tables：描述一段 SQL 依赖的数据来源表，里面存储的是 Hive 表名的列表。

使用 explain dependency 查看 SQL 查询非分区普通表，在 hive cli 中输入以下命令：

```
explain dependency select s_age,count(1) num from student_orc;
```

得到结果：

```
{"input_partitions":[],"input_tables":[{"tablename":"default@student_tb _orc","tabl
etype":"MANAGED_TABLE"}]}
```

使用 explain dependency 查看 SQL 查询分区表，在 hive cli 中输入以下命令：

```
explain dependency select s_age,count(1) num from student_orc_partition;
```

得到结果：

```
{"input_partitions":[{"partitionName":"default@student_orc_partition@ part=0"},
{"partitionName":"default@student_orc_partition@part=1"},
{"partitionName":"default@student_orc_partition@part=2"},
{"partitionName":"default@student_orc_partition@part=3"},
{"partitionName":"default@student_orc_partition@part=4"},
{"partitionName":"default@student_orc_partition@part=5"},
{"partitionName":"default@student_orc_partition@part=6"},
{"partitionName":"default@student_orc_partition@part=7"},
{"partitionName":"default@student_orc_partition@part=8"},
{"partitionName":"default@student_orc_partition@part=9"}],
"input_tables":[{"tablename":"default@student_orc_partition", "tabletype":"MANAGED_
TABLE"}]
```

explain dependency 的使用场景有两个：

- 场景一：快速排除。快速排除因为读取不到相应分区的数据而导致任务数据输出异常。例如，在一个以天分区的任务中，上游任务因为生产过程不可控因素出现异常或者空跑，导致下游任务引发异常。通过这种方式，可以快速查看 SQL 读取的分区是否出现异常。
- 场景二：理清表的输入，帮助理解程序的运行，特别是有助于理解有多重子查询，多表连接的依赖输入。

下面通过两个案例来看 explain dependency 的实际运用：

案例一：识别看似等价的代码

对于刚接触 SQL 的程序员，很容易将

```
select * from a inner join b on a.no=b.no and a.f>1 and a.f<3;
```

等价于

```
select * from a inner join b on a.no=b.no where a.f>1 and a.f<3;
```

我们可以通过案例来查看下它们的区别：

代码 1：

```
select
a.s_no
from student_orc_partition a
inner join
student_orc_partition_only b
on a.s_no=b.s_no and a.part=b.part and a.part>=1 and a.part<=2;
```

代码 2：

```
select
a.s_no
from student_orc_partition a
inner join
student_orc_partition_only b
on a.s_no=b.s_no and a.part=b.part
where a.part>=1 and a.part<=2;
```

我们看下上述两段代码 explain dependency 的输出结果：

代码 1 的 explain dependency 结果：

```
{"input_partitions":
[{"partitionName":"default@student_orc_partition@part=0"},
{"partitionName":"default@student_orc_partition@part=1"},
```

```
{"partitionName":"default@student_orc_partition@part=2"},
{"partitionName":"default@student_orc_partition_only@part=1"},
{"partitionName":"default@student_orc_partition_only@part=2"}],
"input_tables": [{"tablename":"default@student_orc_partition","tabletype":"MANAGED_
TABLE"}, {"tablename":"default@student_orc_partition_only","tabletype":"MANAGED_TAB
LE"}]}
```

代码 2 的 explain dependency 结果：

```
{"input_partitions":
[{"partitionName":"default@student_orc_partition@part=1"},
{"partitionName" : "default@student_orc_partition@part=2"},
{"partitionName" :"default@student_orc_partition_only@part=1"},
{"partitionName":"default@student_orc_partition_only@part=2"}],
"input_tables": [{"tablename":"default@student_orc_partition","tabletype":"MANAGED_
TABLE"}, {"tablename":"default@student_orc_partition_only","tabletype":"MANAGED_TAB
LE"}]}
```

通过上面的输出结果可以看到，其实上述的两个 SQL 并不等价，代码 1 在内连接（inner join）中的连接条件（on）中加入非等值的过滤条件后，并没有将内连接的左右两个表按照过滤条件进行过滤，内连接在执行时会多读取 part=0 的分区数据。而在代码 2 中，会过滤掉不符合条件的分区。

案例二：识别 SQL 读取数据范围的差别

代码 1：

```
explain dependency
select
a.s_no
from student_orc_partition a
left join
student_orc_partition_only b
on a.s_no=b.s_no and a.part=b.part and b.part>=1 and b.part<=2;
```

代码 2：

```
explain dependency
select
a.s_no
from student_orc_partition a
left join
student_orc_partition_only b
on a.s_no=b.s_no and a.part=b.part and a.part>=1 and a.part<=2;
```

以上两个代码的数据读取范围是一样的吗？答案是不一样，我们通过 explain dependency 来看下：

代码 1 的 explain dependency 结果：


```
{"input_partitions":
[{"partitionName": "default@student_orc_partition@part=0"},
{"partitionName":"default@student_orc_partition@part=1"}, …中间省略 7 个分区
{"partitionName":"default@student_orc_partition@part=9"},
{"partitionName":"default@student_orc_partition_only@part=1"},
{"partitionName":"default@student_orc_partition_only@part=2"}],
"input_tables": [{"tablename":"default@student_orc_partition","tabletype":"MANAGED_
TABLE"}, {"tablename":"default@student_orc_partition_only","tabletype":"MANAGED_TAB
LE"}]}
```


代码 2 的 explain dependency 结果：


```
{"input_partitions":
[{"partitionName":"default@student_orc_partition@part=0"},
{"partitionName":"default@student_orc_partition@part=1"}, …中间省略 7 个分区
{"partitionName":"default@student_orc_partition@part=9"},
{"partitionName":"default@student_orc_partition_only@part=0"},
{"partitionName":"default@student_orc_partition_only@part=1"}, …中间省略 7 个分区
{"partitionName":"default@student_orc_partition_only@part=9"}],
"input_tables": [{"tablename":"default@student_orc_partition","tabletype":"MANAGED_
TABLE"}, {"tablename":"default@student_orc_partition_only","tabletype":"MANAGED_TAB
LE"}]}
```


可以看到，对左外连接在连接条件中加入非等值过滤的条件，**如果过滤条件是作用于右表（b 表）有起到过滤的效果，则右表只要扫描两个分区即可，但是左表（a 表）会进行全表扫描。如果过滤条件是针对左表，则完全没有起到过滤的作用，那么两个表将进行全表扫描**。这时的情况就如同全外连接一样都需要对两个数据进行全表扫描。

在使用过程中，容易认为代码片段 2 可以像代码片段 1 一样进行数据过滤，通过查看 explain dependency 的输出结果，可以知道不是如此。

3. explain authorization 的用法

通过 explain authorization 可以知道当前 SQL 访问的数据来源（INPUTS） 和数据输出（OUTPUTS），以及当前 Hive 的访问用户 （CURRENT_USER）和操作（OPERATION）。

在 hive cli 中输入以下命令：

```
explain authorization
select variance(s_score) from student_tb_orc;
```

结果如下：

```
INPUTS:
  default@student_tb_orc
OUTPUTS:
  hdfs://node01:8020/tmp/hive/hdfs/cbf182a5-8258-4157-9194- 90f1475a3ed5/-mr-10000
CURRENT_USER:
  hdfs
OPERATION:
  QUERY
AUTHORIZATION_FAILURES:
  No privilege 'Select' found for inputs { database:default, table:student_ tb_orc,
 columnName:s_score}
```

从上面的信息可知：

上面案例的数据来源是 defalut 数据库中的 student_tb_orc 表；

数据的输出路径是 hdfs://node01:8020/tmp/hive/hdfs/cbf182a5-8258-4157-9194-90f1475a3ed5/-mr-10000；

当前的操作用户是 hdfs，操作是查询；

观察上面的信息我们还会看到 AUTHORIZATION_FAILURES 信息，提示对当前的输入没有查询权限，但如果运行上面的 SQL 的话也能够正常运行。为什么会出现这种情况？**Hive 在默认不配置权限管理的情况下不进行权限验证，所有的用户在 Hive 里面都是超级管理员，即使不对特定的用户进行赋权，也能够正常查询**。

最后

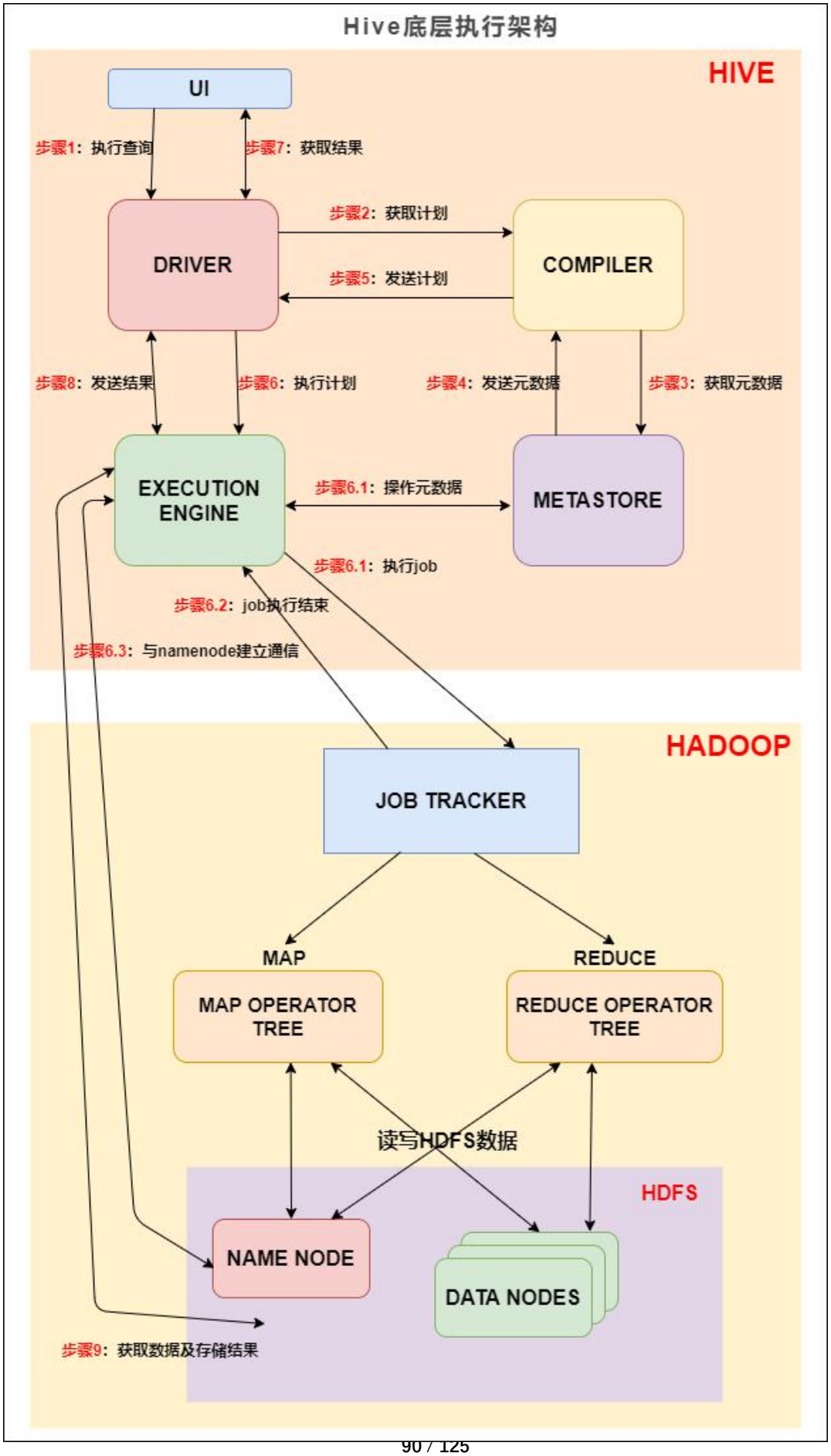
通过上面对 explain 的介绍，可以发现 explain 中有很多值得我们去研究的内容，读懂 explain 的执行计划有利于我们优化 Hive SQL，同时也能提升我们对 SQL 的掌控力。

八、Hive SQL 底层执行原理

本节结构采用宏观着眼，微观入手，从整体到细节的方式剖析 Hive SQL 底层原理。第一节先介绍 Hive 底层的整体执行流程，然后第二节介绍执行流程中的 SQL 编译成 MapReduce 的过程，第三节剖析 SQL 编译成 MapReduce 的具体实现原理。

Hive 底层执行架构

我们先来看下 Hive 的底层执行架构图，Hive 的主要组件与 Hadoop 交互的过程：


Hive底层执行架构
HIVE
UI
步骤1：执行查询
步骤7：获取结果
DRIVER
步骤2：获取计划
步骤5：发送计划
COMPILER
步骤8：发送结果
步骤6：执行计划
步骤4：发送元数据
步骤3：获取元数据
EXECUTION ENGINE
步骤6.1：操作元数据
METASTORE
步骤6.1：执行job
步骤6.2：job执行结束
步骤6.3：与namenode建立通信
HADOOP
JOB TRACKER
MAP
REDUCE
MAP OPERATOR TREE
REDUCE OPERATOR TREE
读写HDFS数据
HDFS
NAME NODE
DATA NODES
步骤9：获取数据及存储结果

Hive 底层执行架构

在 Hive 这一侧，总共有五个组件：

1. UI：用户界面。可看作我们提交 SQL 语句的命令行界面。
2. DRIVER：驱动程序。接收查询的组件。该组件实现了会话句柄的概念。
3. COMPILER：编译器。负责将 SQL 转化为平台可执行的执行计划。对不同的查询块和查询表达式进行语义分析，并最终借助表和从 metastore 查找的分区元数据来生成执行计划。
4. METASTORE：元数据库。存储 Hive 中各种表和分区的所有结构信息。
5. EXECUTION ENGINE：执行引擎。负责提交 COMPILER 阶段编译好的执行计划到不同的平台上。

上图的基本流程是：

步骤 1：UI 调用 DRIVER 的接口；

步骤 2：DRIVER 为查询创建会话句柄，并将查询发送到 COMPILER(编译器)生成执行计划；

步骤 3 和 4：编译器从元数据存储中获取本次查询所需要的元数据，该元数据用于对查询树中的表达式进行类型检查，以及基于查询谓词修建分区；

步骤 5：编译器生成的计划是分阶段的 DAG，每个阶段要么是 map/reduce 作业，要么是一个元数据或者 HDFS 上的操作。将生成的计划发给 DRIVER。

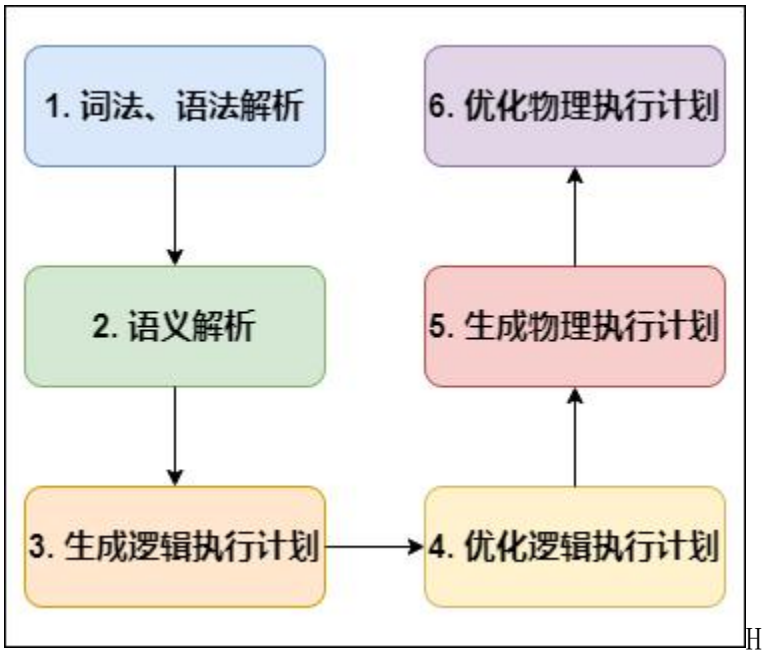
如果是 map/reduce 作业，该计划包括 map operator trees 和一个 reduce operator tree，执行引擎将会把这些作业发送给 MapReduce ：

步骤 6、6.1、6.2 和 6.3：执行引擎将这些阶段提交给适当的组件。在每个 task(mapper/reducer) 中，从 HDFS 文件中读取与表或中间输出相关联的数据，并通过相关算子树传递这些数据。最终这些数据通过序列化器写入到一个临时 HDFS 文件中（如果不需要 reduce 阶段，则在 map 中操作）。临时文件用于向计划中后面的 map/reduce 阶段提供数据。

步骤 7、8 和 9：最终的临时文件将移动到表的位置，确保不读取脏数据(文件重命名在 HDFS 中是原子操作)。对于用户的查询，临时文件的内容由执行引擎直接从 HDFS 读取，然后通过 Driver 发送到 UI。

Hive SQL 编译成 MapReduce 过程

编译 SQL 的任务是在上节中介绍的 COMPILER（编译器组件）中完成的。Hive 将 SQL 转化为 MapReduce 任务，整个编译过程分为六个阶段：



Hive SQL 编译过程

1. 词法、语法解析：Antlr 定义 SQL 的语法规则，完成 SQL 词法，语法解析，将 SQL 转化为抽象语法树 AST Tree；

Antlr 是一种语言识别的工具，可以用来构造领域语言。使用 Antlr 构造特定的语言只需要编写一个语法文件，定义词法和语法替换规则即可，Antlr 完成了词法分析、语法分析、语义分析、中间代码生成的过程。

2. 语义解析：遍历 AST Tree，抽象出查询的基本组成单元 QueryBlock；
3. 生成逻辑执行计划：遍历 QueryBlock，翻译为执行操作树 OperatorTree；
4. 优化逻辑执行计划：逻辑层优化器进行 OperatorTree 变换，合并 Operator，达到减少 MapReduce Job，减少数据传输及 shuffle 数据量；
5. 生成物理执行计划：遍历 OperatorTree，翻译为 MapReduce 任务；
6. 优化物理执行计划：物理层优化器进行 MapReduce 任务的变换，生成最终的执行计划。

下面对这六个阶段详细解析：

为便于理解，我们拿一个简单的查询语句进行展示，对 5 月 23 号的地区维表进行查询：

```
select * from dim.dim_region where dt = '2021-05-23';
```

阶段一：词法、语法解析

根据 Antlr 定义的 sql 语法规则，将相关 sql 进行词法、语法解析，转化为抽象语法树 AST Tree：

```
ABSTRACT SYNTAX TREE:
TOK_QUERY
   TOK_FROM
   TOK_TABREF
         TOK_TABNAME
            dim
               dim_region
   TOK_INSERT
      TOK_DESTINATION
         TOK_DIR
            TOK_TMP_FILE
      TOK_SELECT
         TOK_SELEXPR
            TOK_ALLCOLREF
      TOK_WHERE
         =
            TOK_TABLE_OR_COL
               dt
                  '2021-05-23'
```

阶段二：语义解析

遍历 AST Tree，抽象出查询的基本组成单元 QueryBlock：

AST Tree 生成后由于其复杂度依旧较高，不便于翻译为 mapreduce 程序，需要进行进一步抽象和结构化，形成 QueryBlock。

QueryBlock 是一条 SQL 最基本的组成单元，包括三个部分：输入源，计算过程，输出。简单来讲一个 QueryBlock 就是一个子查询。

QueryBlock 的生成过程为一个递归过程，先序遍历 AST Tree ，遇到不同的 Token 节点(理解为特殊标记)，保存到相应的属性中。

阶段三：生成逻辑执行计划

遍历 QueryBlock，翻译为执行操作树 OperatorTree：

Hive 最终生成的 MapReduce 任务，Map 阶段和 Reduce 阶段均由 OperatorTree 组成。

基本的操作符包括：

- TableScanOperator
- SelectOperator
- FilterOperator

- JoinOperator
- GroupByOperator
- ReduceSinkOperator`

Operator 在 Map Reduce 阶段之间的数据传递都是一个流式的过程。每一个 Operator 对一行数据完成操作后之后将数据传递给 childOperator 计算。

由于 Join/GroupBy/OrderBy 均需要在 Reduce 阶段完成，所以在生成相应操作的 Operator 之前都会先生成一个 ReduceSinkOperator，将字段组合并序列化为 Reduce Key/value, Partition Key。

阶段四：优化逻辑执行计划

Hive 中的逻辑查询优化可以大致分为以下几类：

- 投影修剪
- 推导传递谓词
- 谓词下推
- 将 Select-Select，Filter-Filter 合并为单个操作
- 多路 Join
- 查询重写以适应某些列值的 Join 倾斜

阶段五：生成物理执行计划

生成物理执行计划即是将逻辑执行计划生成的 OperatorTree 转化为 MapReduce Job 的过程，主要分为下面几个阶段：

1. 对输出表生成 MoveTask
2. 从 OperatorTree 的其中一个根节点向下深度优先遍历
3. ReduceSinkOperator 标示 Map/Reduce 的界限，多个 Job 间的界限
4. 遍历其他根节点，遇过碰到 JoinOperator 合并 MapReduceTask
5. 生成 StatTask 更新元数据
6. 剪断 Map 与 Reduce 间的 Operator 的关系

阶段六：优化物理执行计划

Hive 中的物理优化可以大致分为以下几类：

- 分区修剪(Partition Pruning)
- 基于分区和桶的扫描修剪(Scan pruning)
- 如果查询基于抽样，则扫描修剪
- 在某些情况下，在 map 端应用 Group By
- 在 mapper 上执行 Join
- 优化 Union，使 Union 只在 map 端执行
- 在多路 Join 中，根据用户提示决定最后流哪个表
- 删除不必要的 ReduceSinkOperators

- 对于带有 Limit 子句的查询，减少需要为该表扫描的文件数
- 对于带有 Limit 子句的查询，通过限制 ReduceSinkOperator 生成的内容来限制来自 mapper 的输出
- 减少用户提交的 SQL 查询所需的 Tez 作业数量
- 如果是简单的提取查询，避免使用 MapReduce 作业
- 对于带有聚合的简单获取查询，执行不带 MapReduce 任务的聚合
- 重写 Group By 查询使用索引表代替原来的表
- 当表扫描之上的谓词是相等谓词且谓词中的列具有索引时，使用索引扫描

经过以上六个阶段，SQL 就被解析映射成了集群上的 MapReduce 任务。

SQL 编译成 MapReduce 具体原理

在阶段五-生成物理执行计划，即遍历 OperatorTree，翻译为 MapReduce 任务，这个过程具体是怎么转化的呢

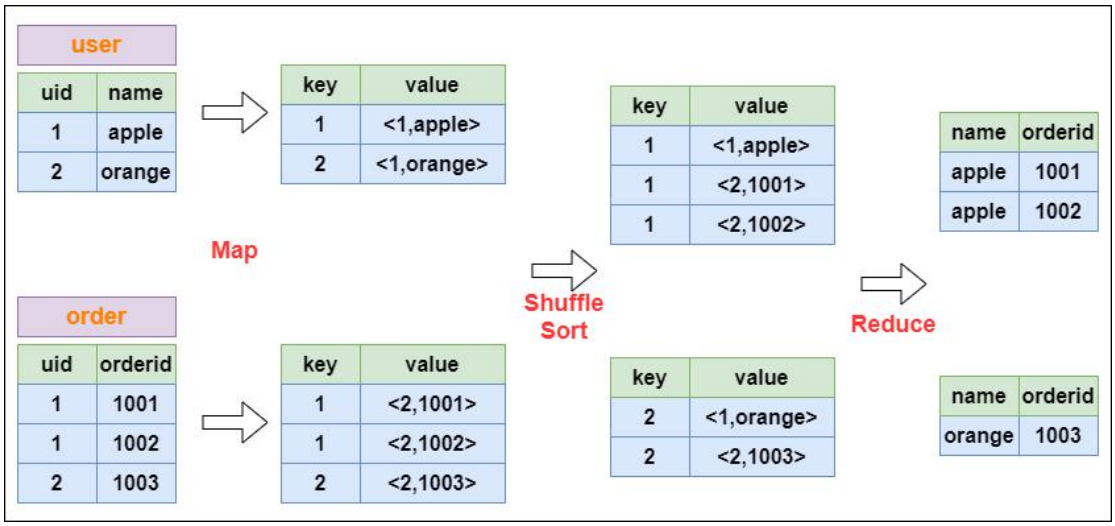
我们接下来举几个常用 SQL 语句转化为 MapReduce 的具体步骤：

Join 的实现原理

以下面这个 SQL 为例，讲解 join 的实现：

```
select u.name, o.orderid from order o join user u on o.uid = u.uid;
```

在 map 的输出 value 中为不同表的数据打上 tag 标记，在 reduce 阶段根据 tag 判断数据来源。MapReduce 的过程如下：



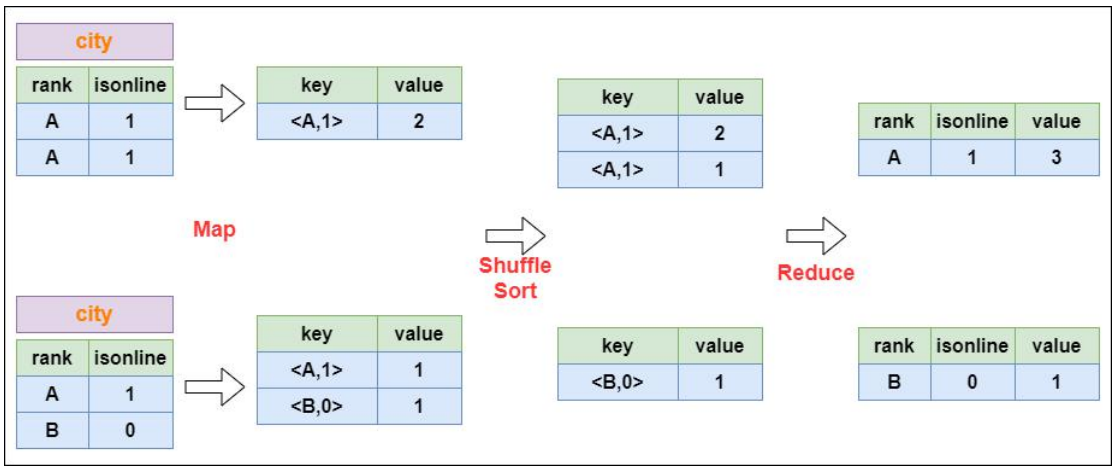
MapReduce CommonJoin 的实现

Group By 的实现原理

以下面这个 SQL 为例，讲解 group by 的实现：

```
select rank, isonline, count(*) from city group by rank, isonline;
```

将 GroupBy 的字段组合为 map 的输出 key 值，利用 MapReduce 的排序，在 reduce 阶段保存 LastKey 区分不同的 key。MapReduce 的过程如下：



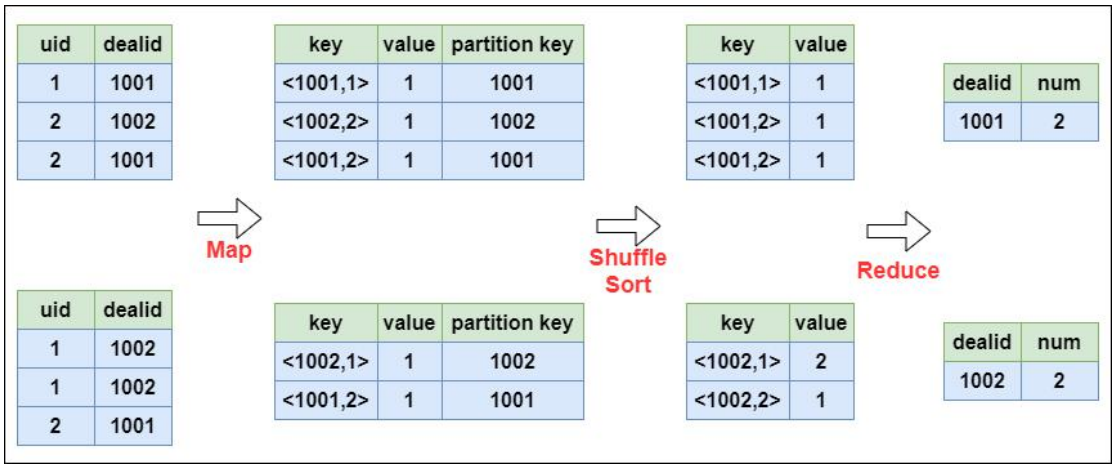
MapReduce Group By 的实现

Distinct 的实现原理

以下面这个 SQL 为例，讲解 distinct 的实现：

```
select dealid, count(distinct uid) num from order group by dealid;
```

当只有一个 distinct 字段时，如果不考虑 Map 阶段的 Hash GroupBy，只需要将 GroupBy 字段和 Distinct 字段组合为 map 输出 key，利用 mapreduce 的排序，同时将 GroupBy 字段作为 reduce 的 key，在 reduce 阶段保存 LastKey 即可完成去重：



MapReduce Distinct 的实现

九、Hive 千亿级数据倾斜

数据倾斜问题剖析

数据倾斜是分布式系统不可避免的问题，任何分布式系统都有几率发生数据倾斜，但有些小伙伴在平时工作中感知不是很明显，这里要注意本篇文章的标题—“千亿级数据”，**为什么说千亿级**，因为如果一个任务的数据量只有几百万，它即使发生了数据倾斜，所有数据都跑到一台机器去执行，对于几百万的数据量，一台机器执行起来还是毫无压力的，这时数据倾斜对我们感知不大，只有数据达到一个量级时，一台机器应付不了这么多的数据，这时如果发生数据倾斜，那么最后就很难算出结果。

所以就需要我们对数据倾斜的问题进行优化，尽量避免或减轻数据倾斜带来的影响。

在解决数据倾斜问题之前，还要再提一句：没有瓶颈时谈论优化，都是自寻烦恼。

大家想想，在 map 和 reduce 两个阶段中，最容易出现数据倾斜的就是 reduce 阶段，因为 map 到 reduce 会经过 shuffle 阶段，在 shuffle 中默认会按照 key 进行 hash，**如果相同的 key 过多，那么 hash 的结果就是大量相同的 key 进入到同一个 reduce 中**，导致数据倾斜。

那么有没有可能在 map 阶段就发生数据倾斜呢，是有这种可能的。

一个任务中，数据文件在进入 map 阶段之前会进行切分，默认是 128M 一个数据块，但是如果**当对文件使用 GZIP 压缩等不支持文件分割操作的压缩方式**时，MR 任务读取压缩后的文件时，是对它切分不了的，该压缩文件只会被一个任务所读取，如果有一个超大的不可切分的压缩文件被一个 map 读取时，就会发生 map 阶段的数据倾斜。

所以，从本质上来说，**发生数据倾斜的原因有两种：一是任务中需要处理大量相同的 key 的数据。二是任务读取不可分割的大文件**。

数据倾斜解决方案

MapReduce 和 Spark 中的数据倾斜解决方案原理都是类似的，以下讨论 Hive 使用 MapReduce 引擎引发的数据倾斜，Spark 数据倾斜也可以此为参照。

1. 空值引发的数据倾斜

实际业务中有些大量的 null 值或者一些无意义的数据参与到计算作业中，表中有大量的 null 值，如果表之间进行 join 操作，就会有 shuffle 产生，这样所有的 null 值都会被分配到一个 reduce 中，必然产生数据倾斜。

之前有小伙伴问，如果 A、B 两表 join 操作，假如 A 表中需要 join 的字段为 null，但是 B 表中需要 join 的字段不为 null，这两个字段根本就 join 不上啊，为什么还会放到一个 reduce 中呢？

这里我们需要明确一个概念，数据放到同一个 reduce 中的原因不是因为字段能不能 join 上，而是因为 shuffle 阶段的 hash 操作，只要 key 的 hash 结果是一样的，它们就会被拉到同一个 reduce 中。

解决方案：

第一种：可以直接不让 null 值参与 join 操作，即不让 null 值有 shuffle 阶段

```
SELECT *
FROM log a
 JOIN users b
 ON a.user_id IS NOT NULL
  AND a.user_id = b.user_id
UNION ALL
SELECT *
FROM log a
WHERE a.user_id IS NULL;
```

第二种：因为 null 值参与 shuffle 时的 hash 结果是一样的，那么我们可以给 null 值随机赋值，这样它们的 hash 结果就不一样，就会进到不同的 reduce 中：

```
SELECT *
FROM log a
 LEFT JOIN users b ON CASE
   WHEN a.user_id IS NULL THEN concat('hive_', rand())
   ELSE a.user_id
  END = b.user_id;
```

2. 不同数据类型引发的数据倾斜

对于两个表 join，表 a 中需要 join 的字段 key 为 int，表 b 中 key 字段既有 string 类型也有 int 类型。当按照 key 进行两个表的 join 操作时，默认的 Hash 操作会按 int 型的 id 来进行分配，这样所有的 string 类型都被分配成同一个 id，结果就是所有的 string 类型的字段进入到一个 reduce 中，引发数据倾斜。

解决方案：

如果 key 字段既有 string 类型也有 int 类型，默认的 hash 就都会按 int 类型来分配，那我们直接把 int 类型都转为 string 就好了，这样 key 字段都为 string，hash 时就按照 string 类型分配了：

```
SELECT *
FROM users a
 LEFT JOIN logs b ON a.usr_id = CAST(b.user_id AS string);
```

3. 不可拆分大文件引发的数据倾斜

当集群的数据量增长到一定规模，有些数据需要归档或者转储，这时候往往会对数据进行压缩；**当对文件使用 GZIP 压缩等不支持文件分割操作的压缩方式，在日后有作业涉及读取压缩后的文件时，该压缩文件只会被一个任务所读取**。如果该压缩文件很大，则处理该文件的 Map 需要花费的时间会远多于读取普通文件的 Map 时间，该 Map 任务会成为作业运行的瓶颈。这种情况也就是 Map 读取文件的数据倾斜。

解决方案：

这种数据倾斜问题没有什么好的解决方案，只能将使用 GZIP 压缩等不支持文件分割的文件转为 bzip 和 zip 等支持文件分割的压缩方式。

所以，**我们在对文件进行压缩时，为避免因不可拆分大文件而引发数据读取的倾斜，在数据压缩的时候可以采用 bzip2 和 Zip 等支持文件分割的压缩算法**。

4. 数据膨胀引发的数据倾斜

在多维聚合计算时，如果进行分组聚合的字段过多，如下：

```
select a，b，c，count（1）from log group by a，b，c with rollup;
```

注：对于最后的 `with rollup` 关键字不知道大家用过没，with rollup 是用来在分组统计数据的基础上再进行统计汇总，即用来得到 group by 的汇总信息。

如果上面的 log 表的数据量很大，并且 Map 端的聚合不能很好地起到数据压缩的情况下，会导致 Map 端产出的数据急速膨胀，这种情况容易导致作业内存溢出的异常。如果 log 表含有数据倾斜 key，会加剧 Shuffle 过程的数据倾斜。

解决方案：

可以拆分上面的 sql，将 `with rollup` 拆分成如下几个 sql：

```
SELECT a, b, c, COUNT(1)
FROM log
GROUP BY a, b, c;
```

```
SELECT a, b, NULL, COUNT(1)
FROM log
GROUP BY a, b;

SELECT a, NULL, NULL, COUNT(1)
FROM log
GROUP BY a;

SELECT NULL, NULL, NULL, COUNT(1)
FROM log;
```

但是，上面这种方式不太好，因为现在是对 3 个字段进行分组聚合，那如果是 5 个或者 10 个字段呢，那么需要拆解的 SQL 语句会更多。

在 Hive 中可以通过参数 `hive.new.job.grouping.set.cardinality` 配置的方式自动控制作业的拆解，该参数默认值是 30。表示针对 grouping sets/rollups/cubes 这类多维聚合的操作，如果最后拆解的键组合大于该值，会启用新的任务去处理大于该值之外的组合。如果在处理数据时，某个分组聚合的列有较大的倾斜，可以适当调小该值。

5. 表连接时引发的数据倾斜

两表进行普通的 repartition join 时，如果表连接的键存在倾斜，那么在 Shuffle 阶段必然会引起数据倾斜。

解决方案：

通常做法是将倾斜的数据存到分布式缓存中，分发到各个 Map 任务所在节点。在 Map 阶段完成 join 操作，即 MapJoin，这避免了 Shuffle，从而避免了数据倾斜。

MapJoin 是 Hive 的一种优化操作，**其适用于小表 JOIN 大表的场景**，由于表的 JOIN 操作是在 Map 端且在内存进行的，所以其并不需要启动 Reduce 任务也就不需要经过 shuffle 阶段，从而能在一定程度上节省资源提高 JOIN 效率。

在 Hive 0.11 版本之前，如果想在 Map 阶段完成 join 操作，必须使用 MAPJOIN 来标记显示地启动该优化操作，**由于其需要将小表加载进内存所以要注意小表的大小**。

如将 a 表放到 Map 端内存中执行，在 Hive 0.11 版本之前需要这样写：

```
select /* +mapjoin(a) */ a.id , a.name, b.age
from a join b
on a.id = b.id;
```

如果想将多个表放到 Map 端内存中，只需在 mapjoin()中写多个表名称即可，用逗号分隔，如将 a 表和 c 表放到 Map 端内存中，则 `/* +mapjoin(a,c) */` 。

在 Hive 0.11 版本及之后，Hive 默认启动该优化，也就是不在需要显示的使用 MAPJOIN 标记，其会在必要的时候触发该优化操作将普通 JOIN 转换成 MapJoin，可以通过以下两个属性来设置该优化的触发时机：

`hive.auto.convert.join=true` 默认值为 true，自动开启 MAPJOIN 优化。

`hive.mapjoin.smalltable.filesize=2500000` 默认值为 2500000(25M)，通过配置该属性来确定使用该优化的表的大小，如果表的大小小于此值就会被加载进内存中。

注意：使用默认启动该优化的方式如果出现莫名其妙的 BUG(比如 MAPJOIN 并不起作用)，就将以下两个属性置为 fase 手动使用 MAPJOIN 标记来启动该优化：

`hive.auto.convert.join=false` (关闭自动 MAPJOIN 转换操作)

`hive.ignore.mapjoin.hint=false` (不忽略 MAPJOIN 标记)

再提一句：将表放到 Map 端内存时，如果节点的内存很大，但还是出现内存溢出的情况，我们可以通过这个参数 `mapreduce.map.memory.mb` 调节 Map 端内存的大小。

6. 确实无法减少数据量引发的数据倾斜

在一些操作中，我们没有办法减少数据量，如在使用 collect_list 函数时：

```
select s_age,collect_list(s_score) list_score
from student
group by s_age
```

collect_list：将分组中的某列转为一个数组返回。

在上述 sql 中，s_age 有数据倾斜，但如果数据量大到一定的数量，会导致处理倾斜的 Reduce 任务产生内存溢出的异常。

collect_list 输出一个数组，中间结果会放到内存中，所以如果 collect_list 聚合太多数据，会导致内存溢出。

有小伙伴说这是 group by 分组引起的数据倾斜，可以开启 `hive.groupby.skewindata` 参数来优化。我们接下来分析下：

开启该配置会将作业拆解成两个作业，第一个作业会尽可能将 Map 的数据平均分配到 Reduce 阶段，并在这个阶段实现数据的预聚合，以减少第二个作业处理的数据量；第二个作业在第一个作业处理的数据基础上进行结果的聚合。

hive.groupby.skewindata 的核心作用在于生成的第一个作业能够有效减少数量。但是对于 collect_list 这类要求全量操作所有数据的中间结果的函数来说，明显起不到作用，反而因为引入新的作业增加了磁盘和网络 I/O 的负担，而导致性能变得更为低下。

解决方案：

这类问题最直接的方式就是调整 reduce 所执行的内存大小。

调整 reduce 的内存大小使用 mapreduce.reduce.memory.mb 这个配置。

总结

通过上面的内容我们发现，**shuffle 阶段堪称性能的杀手**，为什么这么说，一方面 shuffle 阶段是最容易引起数据倾斜的；另一方面 shuffle 的过程中会产生大量的磁盘 I/O、网络 I/O 以及压缩、解压缩、序列化和反序列化等。这些操作都是严重影响性能的。

所以围绕 shuffle 和数据倾斜有很多的调优点：

- Mapper 端的 Buffer 设置为多大？ Buffer 设置得大，可提升性能，减少磁盘 I/O ，但 是对内存有要求，对 GC 有压力； Buffer 设置得小，可能不占用那么多内存， 但是可 能频繁的磁盘 I/O 、频繁的网络 I/O 。

十、Hive 企业级性能优化

Hive 作为大数据平台举足轻重的框架，以其稳定性和简单易用性也成为当前构建企业级数据仓库时使用最多的框架之一。

但是如果我们只局限于会使用 Hive，而不考虑性能问题，就难搭建出一个完美的数仓，所以 Hive 性能调优是我们大数据从业者必须掌握的技能。本节将给大家讲解 Hive 性能调优的一些方法及技巧。

Hive 性能问题排查的方式

当我们发现一条 SQL 语句执行时间过长或者不合理时，我们就要考虑对 SQL 进行优化，优化首先得进行问题排查，那么我们可以通过哪些方式进行排查呢。

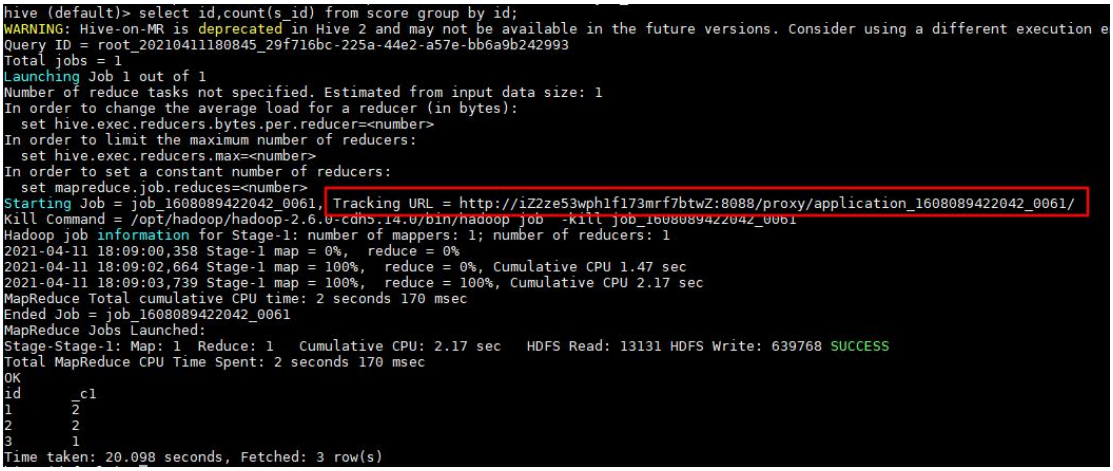
经常使用关系型数据库的同学可能知道关系型数据库的优化的诀窍-**看执行计划**。

如 Oracle 数据库，它有多种类型的执行计划，通过多种执行计划的配合使用，

可以看到根据统计信息推演的执行计划，即 Oracle 推断出来的未真正运行的执行计划；还可以看到实际执行任务的执行计划；能够观察到从数据读取到最终呈现的主要过程和中间的量化数据。可以说，在 Oracle 开发领域，掌握合适的环节，选用不同的执行计划，SQL 调优就不是一件难事。

Hive 中也有执行计划，但是 Hive 的执行计划都是预测的，这点不像 Oracle 和 SQL Server 有真实的计划，可以看到每个阶段的处理数据、消耗的资源和处理的时间等量化数据。Hive 提供的执行计划没有这些数据，这意味着虽然 Hive 的使用者知道整个 SQL 的执行逻辑，但是各阶段耗用的资源状况和整个 SQL 的执行瓶颈在哪里是不清楚的。

想要知道 HiveSQL 所有阶段的运行信息，可以查看 YARN 提供的日志。查看日志的链接，可以在每个作业执行后，在控制台打印的信息中找到。如下图所示：



Hive 提供的执行计划目前可以查看的信息有以下几种：

1. 查看执行计划的基本信息，即 explain；
2. 查看执行计划的扩展信息，即 explain extended；
3. 查看 SQL 数据输入依赖的信息，即 explain dependency；
4. 查看 SQL 操作相关权限的信息，即 explain authorization；
5. 查看 SQL 的向量化描述信息，即 explain vectorization。

在查询语句的 SQL 前面加上关键字 explain 是查看执行计划的基本方法。 用 explain 打开的执行计划包含以下两部分：

- 作业的依赖关系图，即 STAGE DEPENDENCIES；
- 每个作业的详细信息，即 STAGE PLANS。

Hive 中的 explain 执行计划详解可看我之前写的这篇文章：

Hive 底层原理：explain 执行计划详解

注：使用 explain 查看执行计划是 Hive 性能调优中非常重要的一种方式，请务必掌握！

总结：Hive 对 SQL 语句性能问题排查的方式：

1. 使用 explain 查看执行计划；
2. 查看 YARN 提供的日志。

Hive 性能调优的方式

为什么都说性能优化这项工作是比较难的，因为一项技术的优化，必然是一项综合性的工作，它是多门技术的结合。我们如果只局限于一种技术，那么肯定做不好优化的。
下面将从多个完全不同的角度来介绍 Hive 优化的多样性，我们先来一起感受下。

1. SQL 语句优化

SQL 语句优化涉及到的内容太多，因篇幅有限，不能一一介绍到，所以就拿几个典型举例，让大家学到这种思想，以后遇到类似调优问题可以往这几个方面多思考下。

1. union all

```
insert into table stu partition(tp)
select s_age,max(s_birth) stat,'max' tp
from stu_ori
group by s_age

union all

insert into table stu partition(tp)
select s_age,min(s_birth) stat,'min' tp
from stu_ori
group by s_age;
```

我们简单分析上面的 SQl 语句，就是将每个年龄的最大和最小的生日获取出来放到同一张表中，union all 前后的两个语句都是对同一张表按照 s_age 进行分组，然后分别取最大值和最小值。对同一张表相同的字段进行两次分组，这造成了极

大浪费，我们能不能改造下呢，当然是可以的，为大家介绍一个语法： `from ... insert into ...` ，这个语法将 from 前置，作用就是使用一张表，可以进行多次插入操作：

```
--开启动态分区
set hive.exec.dynamic.partition=true;
set hive.exec.dynamic.partition.mode=nonstrict;

from stu_ori

insert into table stu partition(tp)
select s_age,max(s_birth) stat,'max' tp
group by s_age

insert into table stu partition(tp)
select s_age,min(s_birth) stat,'min' tp
group by s_age;
```

上面的 SQL 就可以对 stu_ori 表的 s_age 字段分组一次而进行两次不同的插入操作。

这个例子告诉我们一定要多了解 SQL 语句，如果我们不知道这种语法，一定不会想到这种方式的。

2. distinct

先看一个 SQL，去重计数：

```
select count(1)
from(
  select s_age
  from stu
  group by s_age
) b;
```

这是简单统计年龄的枚举值个数，为什么不用 distinct？

```
select count(distinct s_age)
from stu;
```

有人说因为在数据量特别大的情况下使用第一种方式能够有效避免 Reduce 端的数据倾斜，但是事实如此吗？

我们先不管数据量特别大这个问题，**就当前的业务和环境下使用 distinct 一定会比上面那种子查询的方式效率高**。原因有以下几点：

1. 上面进行去重的字段是年龄字段，要知道年龄的枚举值是非常有限的，就算计算 1 岁到 100 岁之间的年龄，s_age 的最大枚举值才是 100，如果转化成 MapReduce 来解释的话，在 Map 阶段，每个 Map 会对 s_age 去重。由于 s_age 枚举值有限，因而每个 Map 得到的 s_age 也有限，最终得到 reduce 的数据量也就是 map 数量*s_age 枚举值的个数。
2. distinct 的命令会在内存中构建一个 hashtable，查找去重的时间复杂度是 O(1)；group by 在不同版本间变动比较大，有的版本会用构建 hashtable 的形式去重，有的版本会通过排序的方式， 排序最优时间复杂度无法到 O(1)。另外，第一种方式(group by)去重会转化为两个任务，会消耗更多的磁盘网络 I/O 资源。
3. 最新的 Hive 3.0 中新增了 count(distinct) 优化，通过配置 `hive.optimize.countdistinct`，即使真的出现数据倾斜也可以自动优化，自动改变 SQL 执行的逻辑。
4. 第二种方式(distinct)比第一种方式(group by)代码简洁，表达的意思简单明了，如果没有特殊的问题，代码简洁就是优！

这个例子告诉我们，有时候我们不要过度优化，调优讲究适时调优，过早进行调优有可能做的是无用功甚至产生负效应，在调优上投入的工作成本和回报不成正比。调优需要遵循一定的原则。

2. 数据格式优化

Hive 提供了多种数据存储组织格式，不同格式对程序的运行效率也会有极大的影响。

Hive 提供的格式有 TEXT、SequenceFile、RCFile、ORC 和 Parquet 等。

SequenceFile 是一个二进制 key/value 对结构的平面文件，在早期的 Hadoop 平台上被广泛用于 MapReduce 输出/输出格式，以及作为数据存储格式。

Parquet 是一种列式数据存储格式，可以兼容多种计算引擎，如 MapRedcue 和 Spark 等，对多层嵌套的数据结构提供了良好的性能支持，是目前 Hive 生产环境中数据存储的主流选择之一。

ORC 优化是对 RCFile 的一种优化，它提供了一种高效的方式来存储 Hive 数据，同时也能够提高 Hive 的读取、写入和处理数据的性能，能够兼容多种计算引擎。

事实上，在实际的生产环境中，ORC 已经成为了 Hive 在数据存储上的主流选择之一。

我们使用同样数据及 SQL 语句，只是数据存储格式不同，得到如下执行时长：

数据格式	CPU 时间	用户等待耗时
TextFile	33 分	171 秒
SequenceFile	38 分	162 秒
Parquet	2 分 22 秒	50 秒
ORC	1 分 52 秒	56 秒

注：CPU 时间：表示运行程序所占用服务器 CPU 资源的时间。

用户等待耗时：记录的是用户从提交作业到返回结果期间用户等待的所有时间。

查询 TextFile 类型的数据表耗时 33 分钟， 查询 ORC 类型的表耗时 1 分 52 秒，时间得以极大缩短，可见不同的数据存储格式也能给 HiveSQL 性能带来极大的影响。

3. 小文件过多优化

小文件如果过多，对 hive 来说，在进行查询时，每个小文件都会当成一个块，启动一个 Map 任务来完成，而一个 Map 任务启动和初始化的时间远远大于逻辑处理的时间，就会造成很大的资源浪费。而且，同时可执行的 Map 数量是受限的。所以我们有必要对小文件过多进行优化，关于小文件过多的解决的办法，我之前专门写了一篇文章讲解，具体可查看：

解决 hive 小文件过多问题

4. 并行执行优化

Hive 会将一个查询转化成一个或者多个阶段。这样的阶段可以是 MapReduce 阶段、抽样阶段、合并阶段、limit 阶段。或者 Hive 执行过程中可能需要的其他阶段。默认情况下，Hive 一次只会执行一个阶段。不过，某个特定的 job 可能包含众多的阶段，而这些阶段可能并非完全互相依赖的，也就是说有些阶段是可以并行执行的，这样可能使得整个 job 的执行时间缩短。如果有更多的阶段可以并行执行，那么 job 可能就越快完成。

通过设置参数 hive.exec.parallel 值为 true，就可以开启并发执行。在共享集群中，需要注意下，如果 job 中并行阶段增多，那么集群利用率就会增加。

```
set hive.exec.parallel=true; //打开任务并行执行
set hive.exec.parallel.thread.number=16; //同一个 sql 允许最大并行度，默认为 8。
```

当然得是在系统资源比较空闲的时候才有优势，否则没资源，并行也起不来。

5. JVM 优化

JVM 重用是 Hadoop 调优参数的内容，其对 Hive 的性能具有非常大的影响，特别是对于很难避免小文件的场景或 task 特别多的场景，这类场景大多数执行时间都很短。

Hadoop 的默认配置通常是使用派生 JVM 来执行 map 和 Reduce 任务的。这时 JVM 的启动过程可能会造成相当大的开销，尤其是执行的 job 包含有成百上千 task 任务的情况。JVM 重用可以使得 JVM 实例在同一个 job 中重新使用 N 次。N 的值可以在 Hadoop 的 mapred-site.xml 文件中进行配置。通常在 10-20 之间，具体多少需要根据具体业务场景测试得出。

```
<property>
  <name>mapreduce.job.jvm.numtasks</name>
  <value>10</value>
  <description>How many tasks to run per jvm. If set to -1, there is
  no limit.
  </description>
</property>
```

我们也可以在 hive 中设置

```
set  mapred.job.reuse.jvm.num.tasks=10; //这个设置来设置我们的 jvm 重用
```

这个功能的缺点是，开启 JVM 重用将一直占用使用到的 task 插槽，以便进行重用，直到任务完成后才能释放。如果某个“不平衡的”job 中有某几个 reduce task 执行的时间要比其他 Reduce task 消耗的时间多的多的话，那么保留的插槽就会一直空闲着却无法被其他的 job 使用，直到所有的 task 都结束了才会释放。

6. 推测执行优化

在分布式集群环境下，因为程序 Bug（包括 Hadoop 本身的 bug），负载不均衡或者资源分布不均等原因，会造成同一个作业的多个任务之间运行速度不一致，有些任务的运行速度可能明显慢于其他任务（比如一个作业的某个任务进度只有 50%，而其他所有任务已经运行完毕），则这些任务会拖慢作业的整体执行进度。

为了避免这种情况发生，Hadoop 采用了推测执行（Speculative Execution）机制，它根据一定的法则推测出“拖后腿”的任务，并为这样的任务启动一个备份任务，让该任务与原始任务同时处理同一份数据，并最终选用最先成功运行完成任务的计算结果作为最终结果。

设置开启推测执行参数：Hadoop 的 mapred-site.xml 文件中进行配置：

```
<property>
  <name>mapreduce.map.speculative</name>
  <value>true</value>
  <description>If true, then multiple instances of some map tasks
               may be executed in parallel.</description>
</property>

<property>
  <name>mapreduce.reduce.speculative</name>
  <value>true</value>
  <description>If true, then multiple instances of some reduce tasks
               may be executed in parallel.</description>
</property>
```

hive 本身也提供了配置项来控制 reduce-side 的推测执行：

```
set hive.mapred.reduce.tasks.speculative.execution=true
```

关于调优这些推测执行变量，还很难给一个具体的建议。如果用户对于运行时的偏差非常敏感的话，那么可以将这些功能关闭掉。如果用户因为输入数据量很大而需要执行长时间的 map 或者 Reduce task 的话，那么启动推测执行造成的浪费是非常巨大大。

公众号【五分钟学大数据】，大数据领域原创技术号

最后

代码优化原则：

- 理透需求原则，这是优化的根本；
- 把握数据全链路原则，这是优化的脉络；
- 坚持代码的简洁原则，这让优化更加简单；
- 没有瓶颈时谈论优化，这是自寻烦恼。

十一、Hive 大厂面试真题

1. hive 内部表和外部表的区别

未被 external 修饰的是内部表，被 external 修饰的为外部表。

区别：

1. 内部表数据由 Hive 自身管理，外部表数据由 HDFS 管理；
2. 内部表数据存储的位置是 `hive.metastore.warehouse.dir`（默认：`/user/hive/warehouse`），外部表数据的存储位置由自己制定（如果没有 LOCATION，Hive 将在 HDFS 上的`/user/hive/warehouse` 文件夹下以外部表的表名创建一个文件夹，并将属于这个表的数据存放在这里）；
3. **删除内部表会直接删除元数据（metadata）及存储数据；删除外部表仅仅会删除元数据，HDFS 上的文件并不会被删除**。

2. Hive 有索引吗

Hive 支持索引（3.0 版本之前），但是 Hive 的索引与关系型数据库中的索引并不相同，比如，Hive 不支持主键或者外键。并且 Hive 索引提供的功能很有限，效率也并不高，因此 Hive 索引很少使用。

- 索引适用的场景：

适用于不更新的静态字段。以免总是重建索引数据。每次建立、更新数据后，都要重建索引以构建索引表。

- Hive 索引的机制如下：

hive 在指定列上建立索引，会产生一张索引表（Hive 的一张物理表），里面的字段包括：索引列的值、该值对应的 HDFS 文件路径、该值在文件中的偏移量。Hive 0.8 版本后引入 bitmap 索引处理器，这个处理器适用于去重后，值较少的列（例如，某字段的取值只可能是几个枚举值） 因为索引是用空间换时间，索引列的取值过多会导致建立 bitmap 索引表过大。

注意：Hive 中每次有数据时需要及时更新索引，相当于重建一个新表，否则会影响数据查询的效率和准确性，**Hive 官方文档已经明确表示 Hive 的索引不推荐被使用，在新版本的 Hive 中已经被废弃了**。

扩展：Hive 是在 0.7 版本之后支持索引的，在 0.8 版本后引入 bitmap 索引处理器，在 3.0 版本开始移除索引的功能，取而代之的是 2.3 版本开始的物化视图，自动重写的物化视图替代了索引的功能。

3. 运维如何对 hive 进行调度

1. 将 hive 的 sql 定义在脚本当中；
2. 使用 azkaban 或者 oozie 进行任务的调度；
3. 监控任务调度页面。

4. ORC、Parquet 等列式存储的优点

ORC 和 Parquet 都是高性能的存储方式，这两种存储格式总会带来存储和性能上的提升。

Parquet：

1. Parquet 支持嵌套的数据模型，类似于 Protocol Buffers，每一个数据模型的 schema 包含多个字段，每一个字段有三个属性：重复次数、数据类型和字段名。

   重复次数可以是以下三种：required(只出现 1 次)，repeated(出现 0 次或多次)，optional(出现 0 次或 1 次)。每一个字段的数据类型可以分成两种： group(复杂类型)和 primitive(基本类型)。

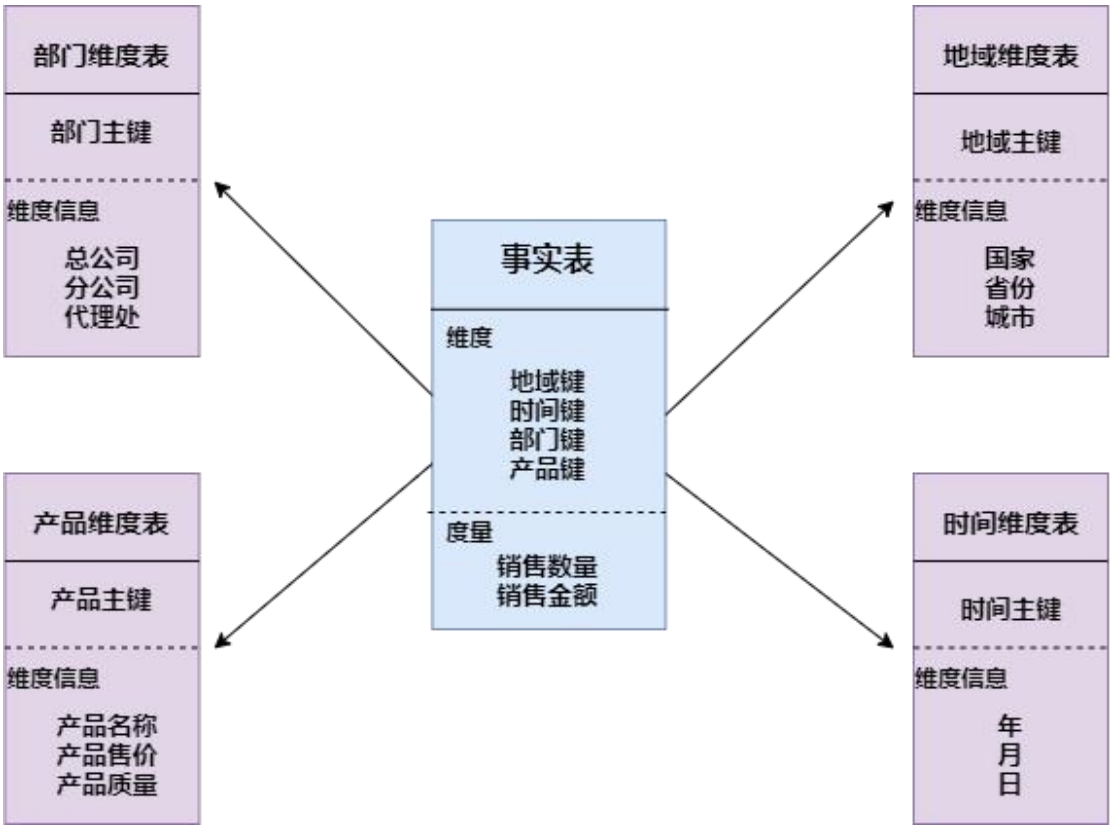
2. Parquet 中没有 Map、Array 这样的复杂数据结构，但是可以通过 repeated 和 group 组合来实现的。
3. 由于 Parquet 支持的数据模型比较松散，可能一条记录中存在比较深的嵌套关系，如果为每一条记录都维护一个类似的树状结可能会占用较大的存储空间，因此 Dremel 论文中提出了一种高效的对于嵌套数据格式的压缩算法：Striping/Assembly 算法。通过 Striping/Assembly 算法，parquet 可以使用较少的存储空间表示复杂的嵌套格式，并且通常 Repetition level 和 Definition level 都是较小的整数值，可以通过 RLE 算法对其进行压缩，进一步降低存储空间。
4. Parquet 文件是以二进制方式存储的，是不可以直接读取和修改的，Parquet 文件是自解析的，文件中包括该文件的数据和元数据。

ORC:

1. ORC 文件是自描述的，它的元数据使用 Protocol Buffers 序列化，并且文件中的数据尽可能的压缩以降低存储空间的消耗。
2. 和 Parquet 类似，ORC 文件也是以二进制方式存储的，所以是不可以直接读取，ORC 文件也是自解析的，它包含许多的元数据，这些元数据都是同构 ProtoBuffer 进行序列化的。
3. ORC 会尽可能合并多个离散的区间尽可能的减少 I/O 次数。
4. ORC 中使用了更加精确的索引信息，使得在读取数据时可以指定从任意一行开始读取，更细粒度的统计信息使得读取 ORC 文件跳过整个 row group，ORC 默认会对任何一块数据和索引信息使用 ZLIB 压缩，因此 ORC 文件占用的存储空间也更小。
5. 在新版本的 ORC 中也加入了对 Bloom Filter 的支持，它可以进一 步提升谓词下推的效率，在 Hive 1.2.0 版本以后也加入了对此的支 持。

5. 数据建模用的哪些模型？

1. 星型模型



星形模式

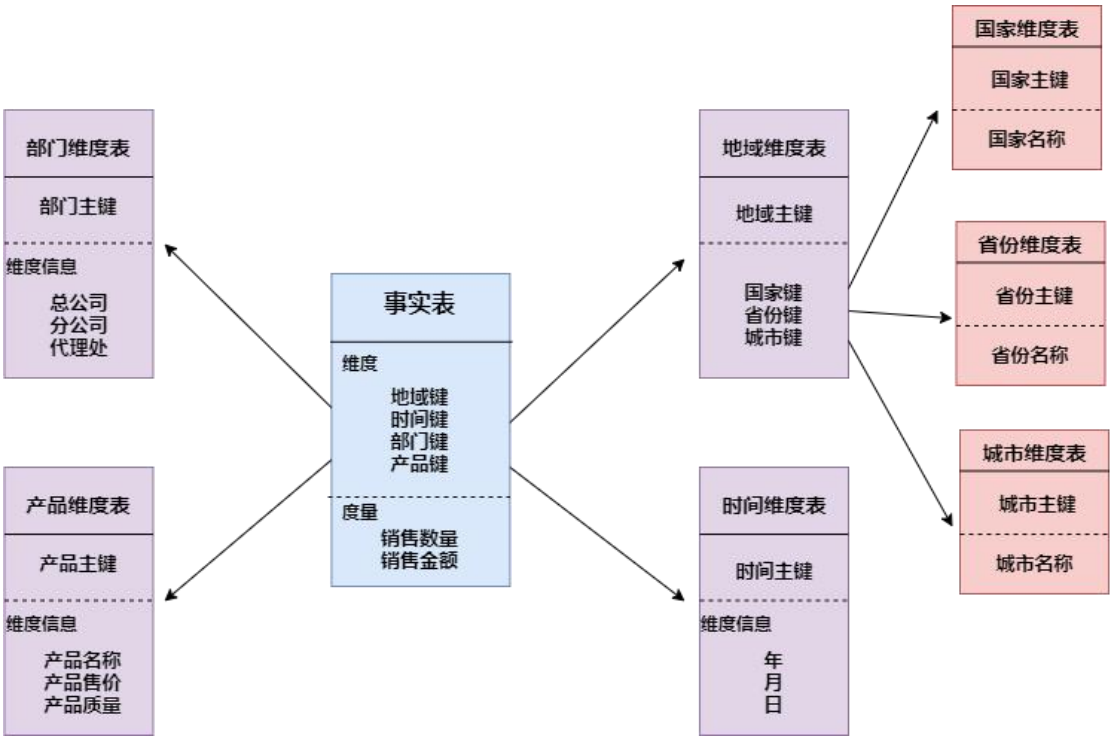
星形模式(Star Schema)是最常用的维度建模方式。星型模式是以事实表为中心，所有的维度表直接连接在事实表上，像星星一样。 星形模式的维度建模由一个事实表和一组维表成，且具有以下特点：

a. 维表只和事实表关联，维表之间没有关联；

b. 每个维表主键为单列，且该主键放置在事实表中，作为两边连接的外键；

c. 以事实表为核心，维表围绕核心呈星形分布。

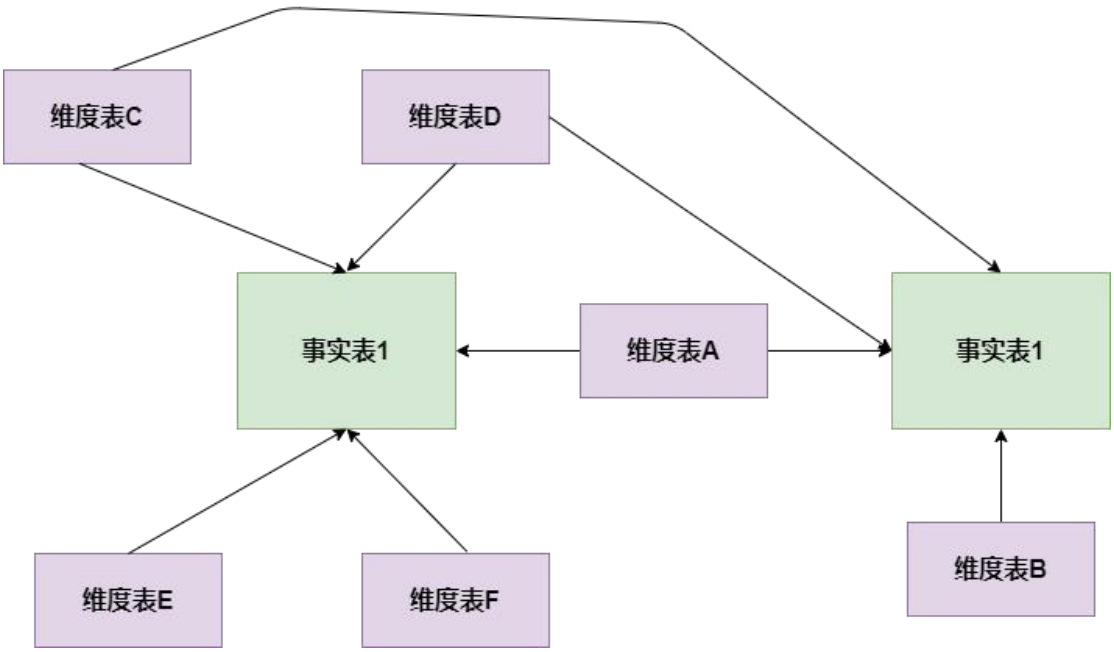
2. 雪花模型



雪花模式

雪花模式(Snowflake Schema)是对星形模式的扩展。**雪花模式的维度表可以拥有其他维度表的**，虽然这种模型相比星型更规范一些，但是由于这种模型不太容易理解，维护成本比较高，而且性能方面需要关联多层维表，性能比星型模型要低。

3. 星座模型



星座模型

星座模式是星型模式延伸而来，星型模式是基于一张事实表的，而**星座模式是基于多张事实表的，而且共享维度信息**。前面介绍的两种维度建模方法都是多维表对应单事实表，但在很多时候维度空间内的事实表不止一个，而一个维表也可能被多个事实表用到。在业务发展后期，绝大部分维度建模都采用的是星座模式。

数仓建模详细介绍可查看：通俗易懂数仓建模

6. 为什么要对数据仓库分层？

- **用空间换时间**，通过大量的预处理来提升应用系统的用户体验（效率），因此数据仓库会存在大量冗余的数据。
- 如果不分层的话，如果源业务系统的业务规则发生变化将会影响整个数据清洗过程，工作量巨大。
- **通过数据分层管理可以简化数据清洗的过程**，因为把原来一步的工作分到了多个步骤去完成，相当于把一个复杂的工作拆成了多个简单的工作，把一个大的黑盒变成了一个白盒，每一层的处理逻辑都相对简单和容易理解，这样我们比较容易保证每一个步骤的正确性，当数据发生错误的时候，往往我们只需要局部调整某个步骤即可。

数据仓库详细介绍可查看：万字详解整个数据仓库建设体系

7. 使用过 Hive 解析 JSON 串吗

Hive 处理 json 数据总体来说有两个方向的路走：

1. 将 json 以字符串的方式整个入 Hive 表，然后通过使用 UDF 函数解析已经导入到 hive 中的数据，比如使用 `LATERAL VIEW json_tuple` 的方法，获取所需要的列名。
2. 在导入之前将 json 拆成各个字段，导入 Hive 表的数据是已经解析过的。这将需要使用第三方的 SerDe。

详细介绍可查看：Hive 解析 Json 数组超全讲解

8. sort by 和 order by 的区别

order by 会对输入做全局排序，因此只有一个 reducer（多个 reducer 无法保证全局有序）只有一个 reducer，会导致当输入规模较大时，需要较长的计算时间。

sort by 不是全局排序，其在数据进入 reducer 前完成排序. 因此，如果用 sort by 进行排序,并且设置 mapred.reduce.tasks>1，则 sort by 只保证每个 reducer 的输出有序，不保证全局有序。

9. 数据倾斜怎么解决

数据倾斜问题主要有以下几种：

1. 空值引发的数据倾斜
2. 不同数据类型引发的数据倾斜
3. 不可拆分大文件引发的数据倾斜
4. 数据膨胀引发的数据倾斜
5. 表连接时引发的数据倾斜
6. 确实无法减少数据量引发的数据倾斜

以上倾斜问题的具体解决方案可查看：Hive 千亿级数据倾斜解决方案

注意：对于 left join 或者 right join 来说，不会对关联的字段自动去除 null 值，对于 inner join 来说，会对关联的字段自动去除 null 值。

小伙伴们在阅读时注意下，在上面的文章（Hive 千亿级数据倾斜解决方案）中，有一处 sql 出现了上述问题（举例的时候原本是想使用 left join 的，结果手误写成了 join）。此问题由公众号读者发现，感谢这位读者指正。

10. Hive 小文件过多怎么解决

1. 使用 hive 自带的 concatenate 命令，自动合并小文件

使用方法：

```
#对于非分区表
alter table A concatenate;

#对于分区表
alter table B partition(day=20201224) concatenate;
```

注意：

1、concatenate 命令只支持 RCFILE 和 ORC 文件类型。

2、使用 concatenate 命令合并小文件时不能指定合并后的文件数量，但可以多次执行该命令。

3、当多次使用 concatenate 后文件数量不在变化，这个跟参数

mapreduce.input.fileinputformat.split.minsize=256mb 的设置有关，可设定每个文件的最小 size。

2. 调整参数减少 Map 数量

设置 map 输入合并小文件的相关参数（执行 Map 前进行小文件合并）：
在 mapper 中将多个文件合成一个 split 作为输入（CombineHiveInputFormat 底层是 Hadoop 的 CombineFileInputFormat 方法）：

```
set hive.input.format=org.apache.hadoop.hive.ql.io.CombineHiveInputFormat; -- 默认
```

每个 Map 最大输入大小（这个值决定了合并后文件的数量）：

```
set mapred.max.split.size=256000000;    -- 256M
```

一个节点上 split 的至少大小（这个值决定了多个 DataNode 上的文件是否需要合并）：

```
set mapred.min.split.size.per.node=100000000;  -- 100M
```

一个交换机下 split 的至少大小(这个值决定了多个交换机上的文件是否需要合并)：

```
set mapred.min.split.size.per.rack=100000000;  -- 100M
```

3. 减少 Reduce 的数量

reduce 的个数决定了输出的文件的个数,所以可以调整 reduce 的个数控制 hive 表的文件数量。
hive 中的分区函数 distribute by 正好是控制 MR 中 partition 分区的，可以通过设置 reduce 的数量，结合分区函数让数据均衡的进入每个 reduce 即可：

```
#设置 reduce 的数量有两种方式，第一种是直接设置 reduce 个数
set mapreduce.job.reduces=10;

#第二种是设置每个 reduce 的大小，Hive 会根据数据总大小猜测确定一个 reduce 个数
set hive.exec.reducers.bytes.per.reducer=5120000000; -- 默认是 1G，设置为 5G

#执行以下语句，将数据均衡的分配到 reduce 中
set mapreduce.job.reduces=10;
insert overwrite table A partition(dt)
select * from B
distribute by rand();
```

对于上述语句解释：如设置 reduce 数量为 10，使用 rand()， 随机生成一个数 x % 10 ， 这样数据就会随机进入 reduce 中，防止出现有的文件过大或过小。

4. 使用 hadoop 的 archive 将小文件归档

Hadoop Archive 简称 HAR，是一个高效地将小文件放入 HDFS 块中的文件存档工具，它能够将多个小文件打包成一个 HAR 文件，这样在减少 namenode 内存使用的同时，仍然允许对文件进行透明的访问。

```
#用来控制归档是否可用
set hive.archive.enabled=true;
#通知Hive 在创建归档时是否可以设置父目录
set hive.archive.har.parentdir.settable=true;
#控制需要归档文件的大小
set har.partfile.size=1099511627776;
```

使用以下命令进行归档：

```
ALTER TABLE A ARCHIVE PARTITION(dt='2021-05-07', hr='12');
```

对已归档的分区恢复为原文件：

```
ALTER TABLE A UNARCHIVE PARTITION(dt='2021-05-07', hr='12');
```

注意：

归档的分区可以查看不能 insert overwrite，必须先 unarchive

Hive 小文件问题具体可查看：解决 hive 小文件过多问题

11. Hive 优化有哪些

1. 数据存储及压缩：

针对 hive 中表的存储格式通常有 orc 和 parquet，压缩格式一般使用 snappy。相比与 textfile 格式表，orc 占有更少的存储。因为 hive 底层使用 MR 计算架构，数据流是 hdfs 到磁盘再到 hdfs，而且会有很多次，所以使用 orc 数据格式和 snappy 压缩策略可以降低 IO 读写，还能降低网络传输量，这样在一定程度上可以节省存储，还能提升 hql 任务执行效率；

2. 通过调参优化：

并行执行，调节 parallel 参数；

调节 jvm 参数，重用 jvm；
设置 map、reduce 的参数；开启 strict mode 模式；
关闭推测执行设置。

3. 有效地减小数据集将大表拆分成子表；结合使用外部表和分区表。

4. SQL 优化

- 大表对大表：尽量减少数据集，可以通过分区表，避免扫描全表或者全字段；
- 大表对小表：设置自动识别小表，将小表放入内存中去执行。

Hive 优化详细剖析可查看：Hive 企业级性能优化

附：九个最易出错的 SQL 讲解

阅读本节小建议：本文适合细嚼慢咽，不要一目十行，不然会错过很多有价值的细节。

在进行数仓搭建和数据分析时最常用的就是 sql，其语法简洁明了，易于理解，目前大数据领域的几大主流框架全部都支持 sql 语法，包括 hive，spark，flink 等，所以 sql 在大数据领域有着不可替代的作用，需要我们重点掌握。
在使用 sql 时如果不熟悉或不仔细，那么在进行查询分析时极容易出错，接下来我们就来看下几个容易出错的 sql 语句及使用注意事项。

1. decimal

hive 除了支持 int,double,string 等常用类型，也支持 decimal 类型，用于在数据库中存储精确的数值，常用在表示金额的字段上

注意事项：

如：decimal(11,2) 代表最多有 11 位数字，其中后 2 位是小数，整数部分是 9 位；

如果**整数部分超过 9 位，则这个字段就会变成 null，如果整数部分不超过 9 位，则原字段显示**；

如果**小数部分不足 2 位，则后面用 0 补齐两位，如果小数部分超过两位，则超出部分四舍五入**；

也可直接写 decimal，后面不指定位数，默认是 decimal(10,0) 整数 10 位，没有小数

2. location

```
表创建的时候可以用 location 指定一个文件或者文件夹
create  table stu(id int ,name string)  location '/user/stu2';
```

注意事项：

创建表时使用 location， 当**指定文件夹时，hive 会加载文件夹下的所有文件，当表中无分区时，这个文件夹下不能再有文件夹，否则报错。**

当表是分区表时，比如 partitioned by (day string)， 则这个文件夹下的每一个文件夹就是一个分区，且文件夹名为 day=20201123 这种格式，然后使用：msck repair table score; 修复表结构，成功之后即可看到数据已经全部加载到表当中去了

3. load data 和 load data local

```
从 hdfs 上加载文件
load data inpath '/hivedatas/techer.csv' into table techer;

从本地系统加载文件
load data local inpath '/user/test/techer.csv' into table techer;
```

注意事项：

1. 使用 load data local 表示**从本地文件系统加载，文件会拷贝到 hdfs 上**
2. 使用 load data 表示**从 hdfs 文件系统加载，文件会直接移动到 hive 相关目录下**，注意不是拷贝过去，因为 hive 认为 hdfs 文件已经有 3 副本了，没必要再次拷贝了
3. 如果表是分区表，load 时不指定分区会报错
4. 如果加载相同文件名的文件，会被自动重命名

4. drop 和 truncate

```
删除表操作
drop table score1;
```

```
清空表操作
truncate table score2;
```

注意事项：

如果 hdfs 开启了回收站，drop 删除的表数据是可以从回收站恢复的，表结构恢复不了，需要自己重新创建；truncate 清空的表是不进回收站的，所以无法恢复 truncate 清空的表。

所以 truncate 一定慎用，一旦清空除物理恢复外将无力回天

5. join 连接

```
INNER JOIN 内连接：只有进行连接的两个表中都存在与连接条件相匹配的数据才会被保留下来
select * from techer t [inner] join course c on t.t_id = c.t_id; -- inner 可省略

LEFT OUTER JOIN 左外连接：左边所有数据会被返回，右边符合条件的被返回
select * from techer t left join course c on t.t_id = c.t_id; -- outer 可省略

RIGHT OUTER JOIN 右外连接：右边所有数据会被返回，左边符合条件的被返回、
select * from techer t right join course c on t.t_id = c.t_id;

FULL OUTER JOIN 满外(全外)连接：将会返回所有表中符合条件的所有记录。如果任一表的指定字段没有符合条件的值的话，那么就使用 NULL 值替代。
SELECT * FROM techer t FULL JOIN course c ON t.t_id = c.t_id ;
```

注意事项：

1. hive2 版本已经支持不等值连接，就是 join on 条件后面可以使用大于小于符号;并且也支持 join on 条件后跟 or (早前版本 on 后只支持 = 和 and，不支持 > < 和 or)
2. 如 hive 执行引擎使用 MapReduce，一个 join 就会启动一个 job，一条 sql 语句中如有多个 join，则会启动多个 job

注意：表之间用逗号(,)连接和 inner join 是一样的，例：

```
select tableA.id, tableB.name from tableA , tableB where tableA.id=tableB.id;
和
select tableA.id, tableB.name from tableA join tableB on tableA.id=tableB.id;
```

它们的执行效率没有区别，只是书写方式不同，用逗号是 sql 89 标准，join 是 sql 92 标准。用逗号连接后面过滤条件用 where ，用 join 连接后面过滤条件是 on。

6. left semi join

为什么把这个单独拿出来说，因为它和其他的 join 语句不太一样，
这个语句的作用和 in/exists 作用是一样的，是 in/exists 更高效的实现

```
SELECT A.* FROM A where id in (select id from B)
```

```
SELECT A.* FROM A left semi join B ON A.id=B.id
```

上述两个 sql 语句执行结果完全一样，只不过第二个执行效率高

注意事项：

1. left semi join 的限制是：join 子句中右边的表**只能在 on 子句中设置过滤条件**，在 where 子句、select 子句或其他地方过滤都不行。
2. left semi join 中 on 后面的过滤条件**只能是等于号**，不能是其他的。
3. left semi join 是只传递表的 join key 给 map 阶段，因此 left semi join 中最后 select 的**结果只许出现左表**。
4. 因为 left semi join 是 in(keySet) 的关系，遇到**右表重复记录，左表会跳过**

7. 聚合函数中 null 值

hive 支持 count(),max(),min(),sum(),avg() 等常用的聚合函数

注意事项：

聚合操作时要注意 null 值：

count(*) 包含 null 值，统计所有行数；

count(id) 不包含 id 为 null 的值；

min **求最小值是不包含 null**，除非所有值都是 null；

avg **求平均值也是不包含 null**。

以上需要特别注意，null 值最容易导致算出错误的结果

8. 运算符中 null 值

hive 中支持常用的算术运算符(+,-,*,/)
比较运算符(>, <, =)
逻辑运算符(in, not in)

以上运算符计算时要特别注意 null 值

注意事项：

1. **每行中的列字段相加或相减，如果含有 null 值，则结果为 null**
   例：有一张商品表（product）

id	price	dis_amount
1	100	20
2	120	null

各字段含义： id（商品 id）、price（价格）、dis_amount（优惠金额）

我想算**每个商品优惠后实际的价格**，sql 如下：

```
select id, price - dis_amount as real_amount from product;
```

得到结果如下：

id	real_amount
1	80
2	null

id=2 的商品价格为 null，结果是错误的。

我们可以**对 null 值进行处理**，sql 如下：

```
select id, price - coalesce(dis_amount,0) as real_amount from product;
```

```
使用 coalesce 函数进行 null 值处理下，得到的结果就是准确的
```

```
coalesce 函数是返回第一个不为空的值
如上 sql：如果 dis_amount 不为空，则返回 dis_amount，如果为空，则返回 0
```

2. **小于是不包含 null 值**，如 id < 10；是不包含 id 为 null 值的。
3. **not in 是不包含 null 值的**，如 city not in ('北京','上海')，这个条件得出的结果是 city 中不包含 北京，上海和 null 的城市。

9. and 和 or

在 sql 语句的过滤条件或运算中，如果有多个条件或多个运算，我们都会考虑优先级，如乘除优先级高于加减，乘除或者加减它们之间优先级平等，谁在前就先

算谁。那 and 和 or 呢，看似 and 和 or 优先级平等，谁在前先算谁，但是，**and 的优先级高于 or**。

注意事项：

例：

还是一张商品表（product）

id	classify	price
1	电器	70
2	电器	130
3	电器	80
4	家具	150
5	家具	60
6	食品	120

我想要统计下电器或者家具这两类中价格大于 100 的商品，sql 如下：

```
select * from product where classify = '电器' or classify = '家具' and price>100
```

得到结果

id	classify	price
1	电器	70
2	电器	130
3	电器	80
4	家具	150

结果是错误的，把所有的电器类型都查询出来了，原因就是 and 优先级高于 or，上面的 sql 语句实际执行的是，先找出 classify = ’家具’ and price>100 的，然后在找出 classify = ’电器’ 的

正确的 sql 就是加个括号，先计算括号里面的：

```
select * from product where (classify = '电器' or classify = '家具') and price>100
```

最后

第一时间获取最新大数据技术，尽在公众号：五分钟学大数据

搜索公众号：五分钟学大数据，学更多大数据技术！

可直接扫码关注