# OLAP和OLTP的区别：

联机分析处理和联机事务处理。

OLAP是数据仓库系统的主要应用，支持复杂的分析操作，侧重决策支持，提供直观易懂的查询结果。

OLTP传统关系型数据库主要应用，主要是基本的、日常的事务处理，例如银行交易。

区别：

吞吐量：OLTP更大

Sql语句类型：OLAP主要是查询和批量DML，OLTP主要是插入和修改

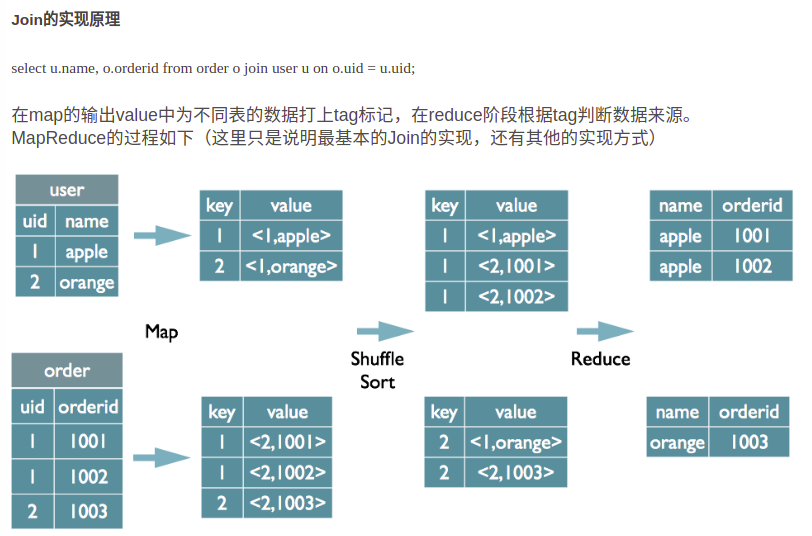
索引类型：OLAP主要是Bitmap索引，OLTP主要是B\*索引

访问方式：OLAP全表扫描，OLTP索引访问

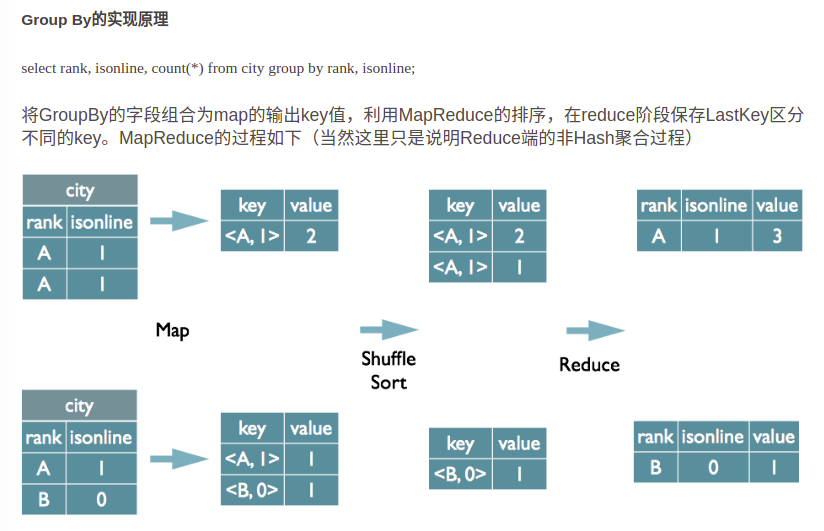
连接方式：OLAP hash join， OLTP Nested\_loop

物化视图：OLAP大量使用，OLTP使用不多

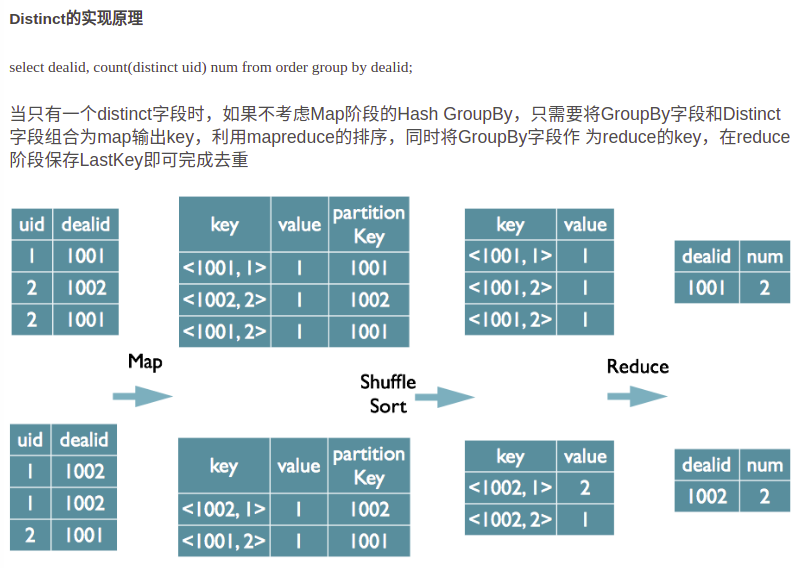
# MapReduce实现SQL操作的原理：



将join key作为key值，也可以认为是当做分区字段，然后相同key值的会分到一起，因为在value中已经进行了tag标记，然后对相同key值的进行Join即可。



首先将city表切片，发给不同的map端进行并行处理，将分组字段作为key，也同时作为分区键，通过shuffle sort就能够把所有相同的分区的分在一起，然后经过reduce处理，就能得到分组字段的值和分组字段的聚合函数表示的值。



分区键还是分组键，只不过key变成了分组键加distinct键。

# HiveSql解析原理:

* 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
* 遍历AST Tree，抽象出查询的基本组成单元QueryBlock
* 遍历QueryBlock，翻译为执行操作树OperatorTree
* 逻辑层优化器进行OperatorTree变换，合并不必要的ReduceSinkOperator，减少shuffle数据量
* 遍历OperatorTree，翻译为MapReduce任务
* 物理层优化器进行MapReduce任务的变换，生成最终的执行计划

**Antlr**

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。

Hive中语法规则的定义文件在0.10版本以前是Hive.g一个文件，随着语法规则越来越复杂，由语法规则生成的Java解析类可能超过Java类文 件的最大上限，0.11版本将Hive.g拆成了5个文件，词法规则HiveLexer.g和语法规则的4个文件 SelectClauseParser.g，FromClauseParser.g，IdentifiersParser.g，HiveParser.g。

**样例hive SQL**

**FROM**

(

**SELECT**

    p.datekey datekey,

    p.userid userid,

    c.clienttype

**FROM**

    detail.usersequence\_client c

    JOIN fact.orderpayment p **ON** p.orderid = c.orderid

    JOIN **default**.user du **ON** du.userid = p.userid

**WHERE** p.datekey = 20131118

) base

**INSERT** OVERWRITE **TABLE** `test`.`customer\_kpi`

**SELECT**

  base.datekey,

  base.clienttype,

  count(**distinct** base.userid) buyer\_count

**GROUP** **BY**  base.datekey, base.clienttype

## Phase1 SQL生成AST Tree

Antlr对Hive SQL解析的代码如下，HiveLexerX，HiveParser分别是Antlr对语法文件Hive.g编译后自动生成的词法解析和语法解析类，在这两个类中进行复杂的解析。

HiveLexerX lexer = new HiveLexerX(new ANTLRNoCaseStringStream(command));    //词法解析，忽略关键词的大小写

TokenRewriteStream tokens = new TokenRewriteStream(lexer);

if (ctx != null) {

   ctx.setTokenRewriteStream(tokens);

}

HiveParser parser = new HiveParser(tokens);                                 //语法解析

parser.setTreeAdaptor(adaptor);

HiveParser.statement\_return r = null;

try {

   r = parser.statement();                                                   //转化为AST Tree

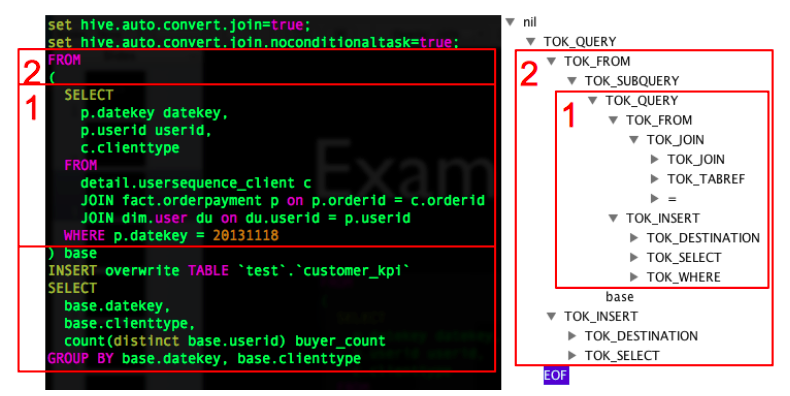
} catch (RecognitionException e) {

   e.printStackTrace();

  throw new ParseException(parser.errors);

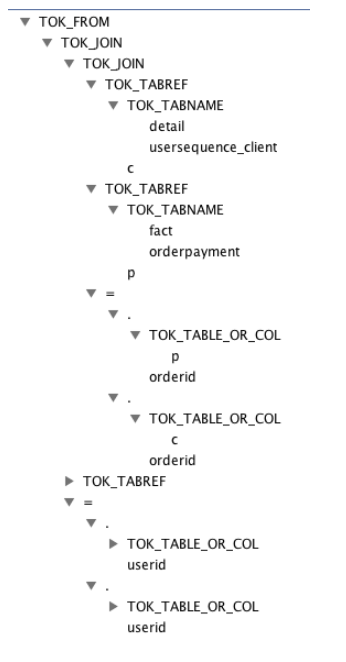
}

最终生成的AST Tree如下图右侧（使用Antlr Works生成，Antlr Works是Antlr提供的编写语法文件的编辑器），图中只是展开了骨架的几个节点，没有完全展开。



这里注意一下内层子查询也会生成一个TOK\_DESTINATION节点。请看上面SelectStatement的语法规则，这个节点是在语法改写中特 意增加了的一个节点。原因是Hive中所有查询的数据均会保存在HDFS临时的文件中，无论是中间的子查询还是查询最终的结果，Insert语句最终会将数据写入表所在的HDFS目录下。

详细来看，将内存子查询的from子句展开后，得到如下AST Tree，每个表生成一个TOK\_TABREF节点，Join条件生成一个“=”节点。其他SQL部分类似，不一一详述。



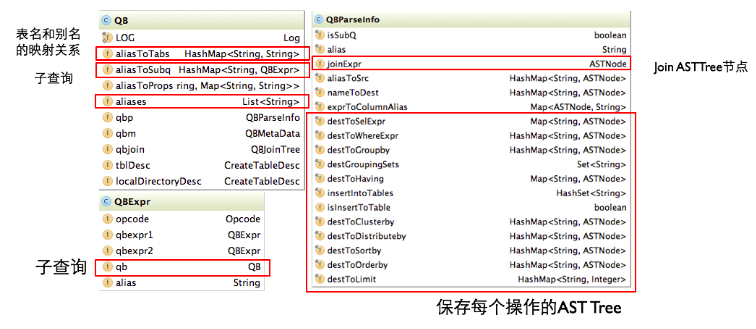
**SQL基本组成单元QueryBlock**

AST Tree仍然非常复杂，不够结构化，不方便直接翻译为MapReduce[程序](http://www.xuebuyuan.com/" \t "/home/vincent/Documents\\x/_blank)，AST Tree转化为QueryBlock就是将SQL进一部抽象和结构化。

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。

下图为Hive中QueryBlock相关对象的类图，解释图中几个重要的属性

* QB#aliasToSubq（表示QB类的aliasToSubq属性）保存子查询的QB对象，aliasToSubq key值是子查询的别名
* QB#qbp 即QBParseInfo保存一个基本SQL单元中的给个操作部分的AST Tree结构，
* QBParseInfo#nameToDest这个HashMap保存查询单元的输出，key的形式是inclause-i（由于Hive 支持Multi Insert语句，所以可能有多个输出），value是对应的ASTNode节点，即TOK\_DESTINATION节点。类QBParseInfo其余 HashMap属性分别保存输出和各个操作的ASTNode节点的对应关系。
* QBParseInfo#JoinExpr保存TOK\_JOIN节点。
* QB#QBJoinTree是对Join语法树的结构化。
* QB#qbm保存每个输入表的元信息，比如表在HDFS上的路径，保存表数据的文件格式等。
* QBExpr这个对象是为了表示Union操作。



## Phase2 AST Tree生成QueryBlock

AST Tree生成QueryBlock的过程是一个递归的过程，先序遍历AST Tree，遇到不同的Token节点，保存到相应的属性中，主要包含以下几个过程

* TOK\_QUERY => 创建QB对象，循环递归子节点
* TOK\_FROM => 将表名语法部分保存到QB对象的TOK\_INSERT => 循环递归子节点
* TOK\_DESTINATION => 将输出目标的语法部分保存在QBParseInfo对象的nameToDest属性中
* TOK\_SELECT => 分别将查询表达式的语法部分保存在destToAggregationExprs、TOK\_WHERE => 将Where部分的语法保存在QBParseInfo对象的destToWhereExpr属性中

最终样例SQL生成两个QB对象，QB对象的关系如下，QB1是外层查询，QB2是子查询

QB1 \ QB2

**逻辑操作符Operator**

Hive最终生成的MapReduce任务，Map阶段和Reduce阶段均由OperatorTree组成。逻辑操作符，就是在Map阶段或者Reduce阶段完成单一特定的操作。

基本的操作符包括TableScanOperator，SelectOperator，FilterOperator，JoinOperator，GroupByOperator，ReduceSinkOperator

从名字就能猜出各个操作符完成的功能，TableScanOperator从MapReduce框架的Map接口原始输入表的数据，控制扫描表的数据行数，标记是从原表中取数据。JoinOperator完成Join操作。FilterOperator完成过滤操作

ReduceSinkOperator将Map端的字段组合序列化为Reduce Key/value, Partition Key，只可能出现在Map阶段，同时也标志着Hive生成的MapReduce程序中Map阶段的结束。

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。

## Phase3 QueryBlock生成Operator Tree

QueryBlock生成Operator Tree就是遍历上一个过程中生成的QB和QBParseInfo对象的保存语法的属性，包含如下几个步骤：

* QB#aliasToSubq => 有子查询，递归调用
* QB#aliasToTabs => TableScanOperator
* QBParseInfo#joinExpr => QBJoinTree => ReduceSinkOperator + JoinOperator
* QBParseInfo#destToWhereExpr => FilterOperator
* QBParseInfo#destToGroupby => ReduceSinkOperator + GroupByOperator
* QBParseInfo#destToOrderby => ReduceSinkOperator + ExtractOperator

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。

**样例hive SQL**

**FROM**

(

**SELECT**

    p.datekey datekey,

    p.userid userid,

    c.clienttype

**FROM**

    detail.usersequence\_client c

    JOIN fact.orderpayment p **ON** p.orderid = c.orderid

    JOIN **default**.user du **ON** du.userid = p.userid

**WHERE** p.datekey = 20131118

) base

**INSERT** OVERWRITE **TABLE** `test`.`customer\_kpi`

**SELECT**

  base.datekey,

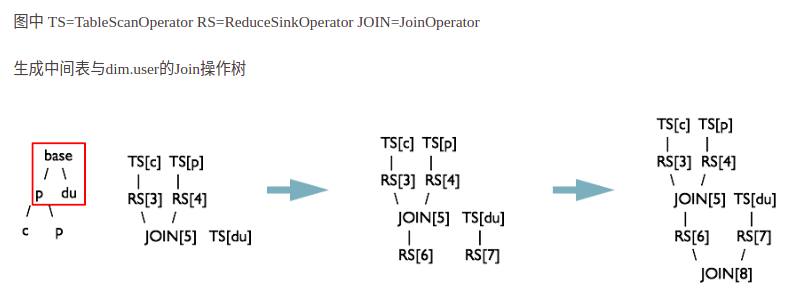
  base.clienttype,

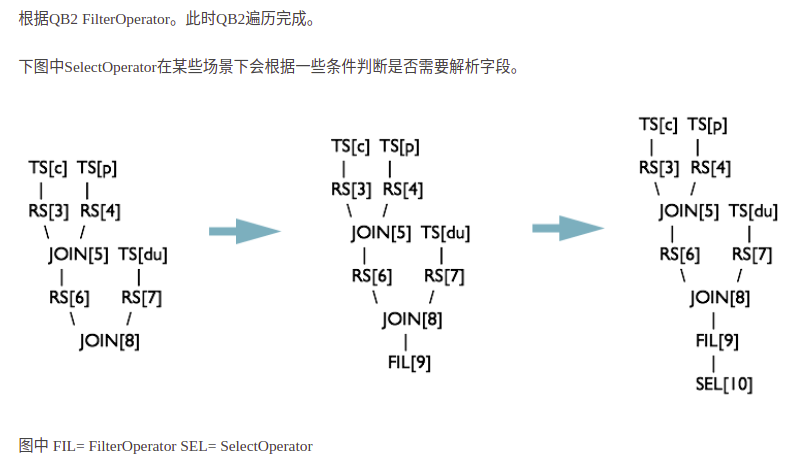
  count(**distinct** base.userid) buyer\_count

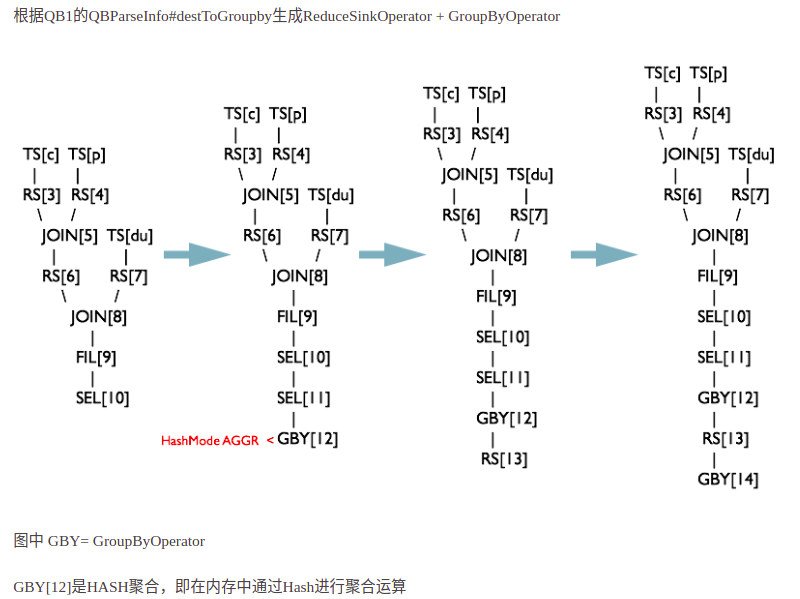
**GROUP** **BY**  base.datekey, base.clienttype

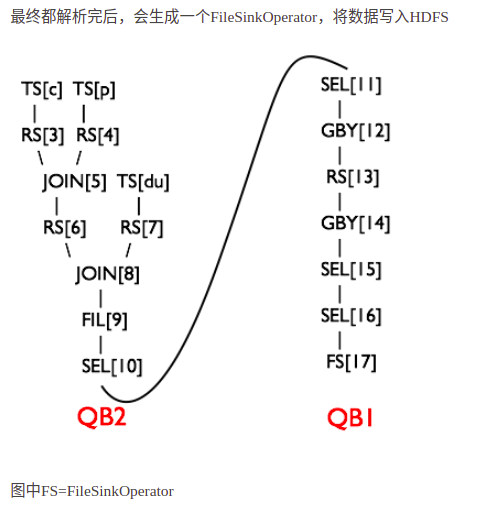
先序遍历QBJoinTree, 类QBJoinTree保存了左右表的ASTNode和这个查询的别名。







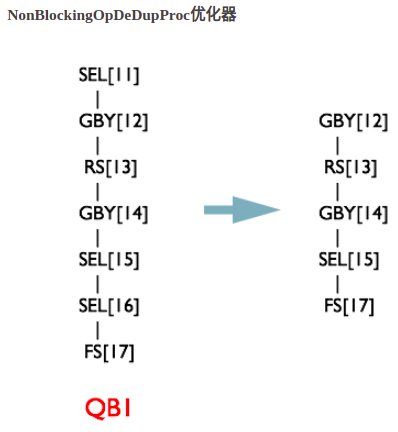
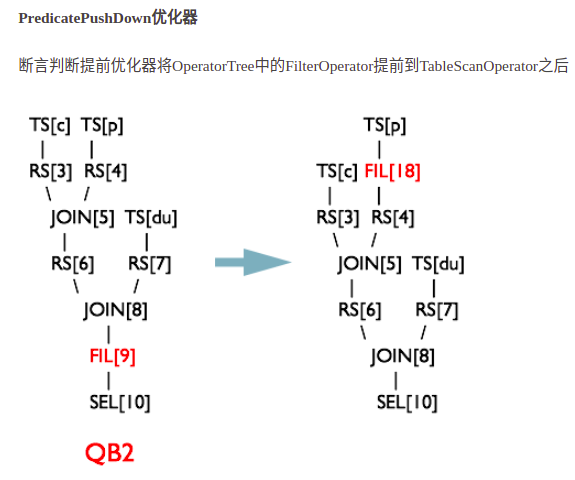




## Phase4 逻辑层优化器

大部分逻辑层优化器通过变换OperatorTree，合并操作符，达到减少MapReduce Job，减少shuffle数据量的目的。

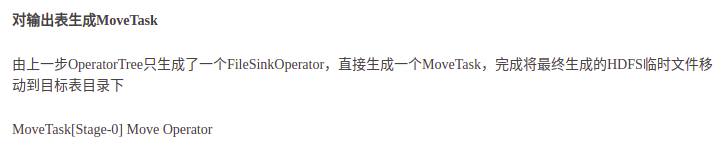
对于样例SQL，有两个优化器对其进行优化。下面分别介绍这两个优化器的作用，并补充一个优化器ReduceSinkDeDuplication的作用

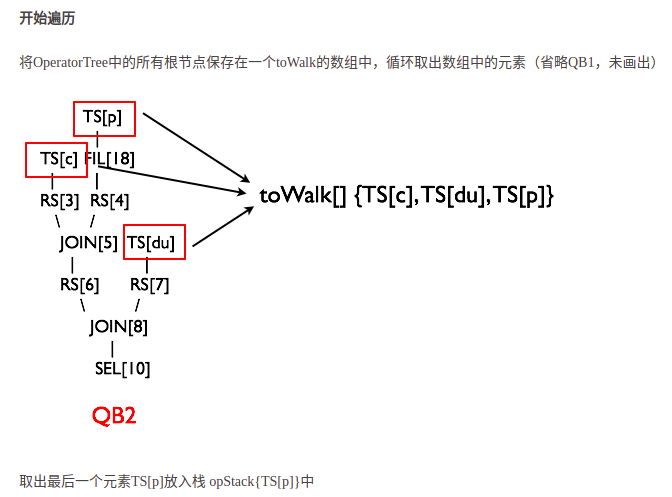


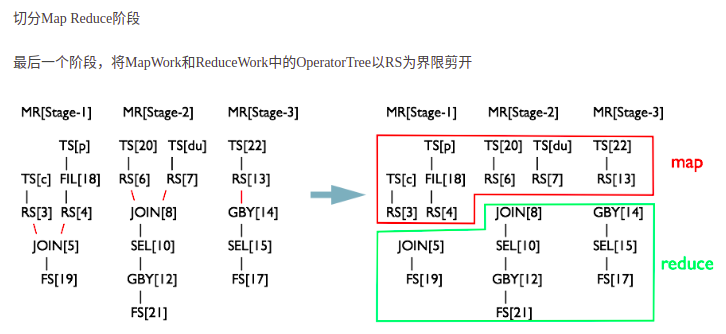
## Phase5 OperatorTree生成MapReduce Job的过程

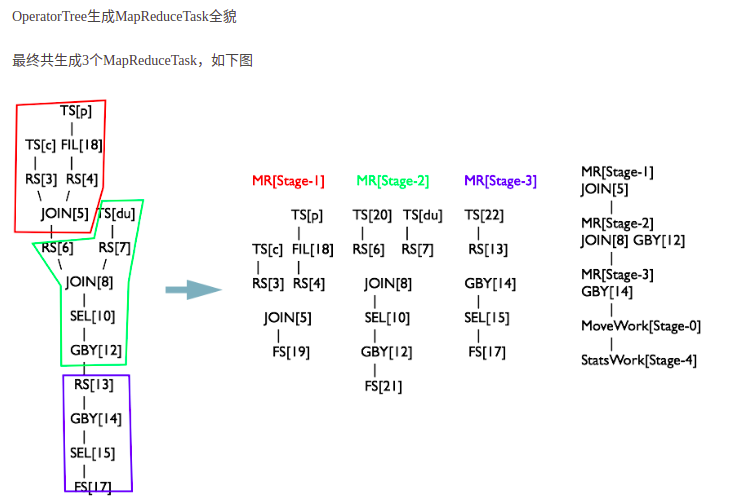
OperatorTree转化为MapReduce Job的过程分为下面几个阶段

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

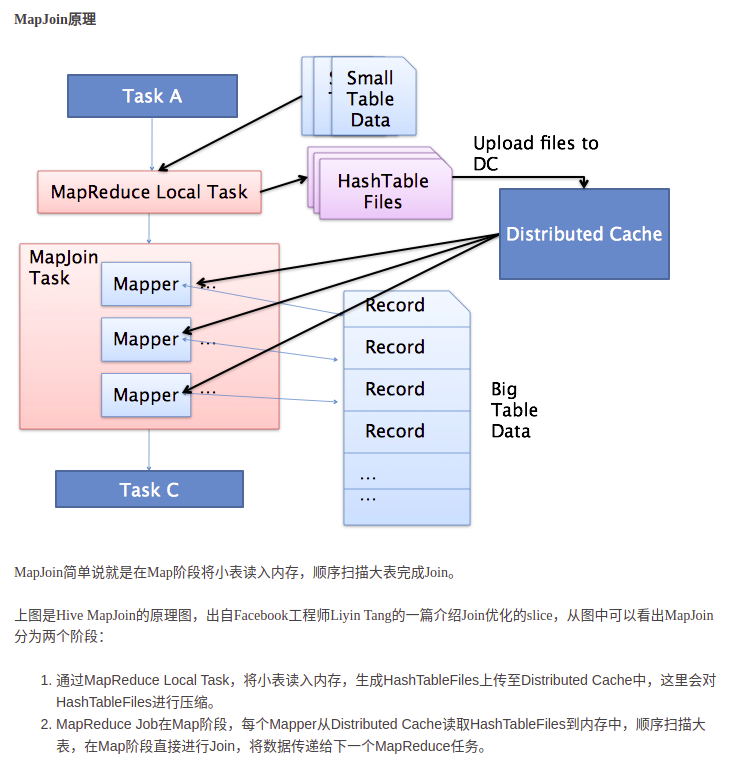








## Phase6 物理层优化器



# 4、Hive和HBase的区别

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，并提供简单的sql查询功能，可以将sql语句转换为MapReduce任务进行运行。

Hive底层是HDFS，以目录的格式存在，存在HDFS的一般是内表，删除时既删除元数据也删除表数据，而映射HBase中的四维表的数据一般是外表，删除表时只会删除元数据，而不会删除表数据。

HBase是Hadoop的数据库，一个分布式、可扩展、大数据的存储

# 5、行存数据库和列存数据库

**行存的缺点**

行式数据库在读取数据的时候会存在一个固有的“缺陷”比如，所选择查询的目标即使只涉及少数几项属性但由于这些目标数据埋藏在各行数据单元中而行单元往往又特别大应用程序必须读取每一条完整的行记录从而使得读取效率大大降低对此，行式数据库给出的优化方案是加“索引” 和 物化视图， 无法从根本上解决查询性能和维护成本等问题也不适用于数据仓库等应用场景。

**列存的优点**

因为各列独立存储，且数据类型已知，可以针对该列的数据类型、数据量大小等因素动态选择压缩算法，以提高物理存储利用率;如果某一行的某一列没有数据，那在列存储时，就可以不存储该列的值，这将比行式存储更节省空间。

**列存的缺点**

数据需要频繁更新的交易场景

表中列属性较少的小量数据库场景

不适合做含有删除和更新的实时操作

**行存列存一统**

随着列式数据库的发展，传统的行式数据库加入了列式存储的支持，形成具有两种存储方式的数据库系统。

# 数据仓库相关知识

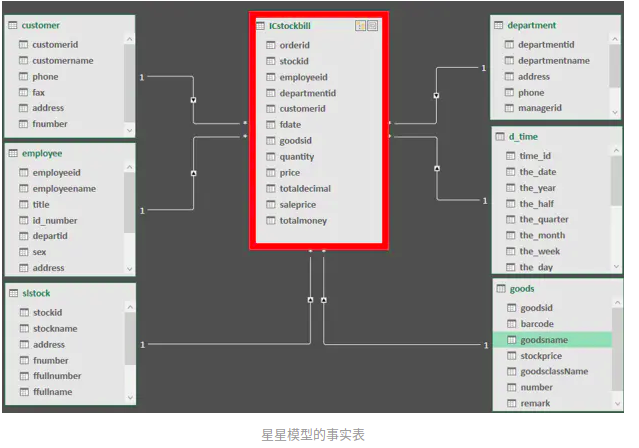
## 数据仓库特点

面向主题、集成的、不易失的和时间变化

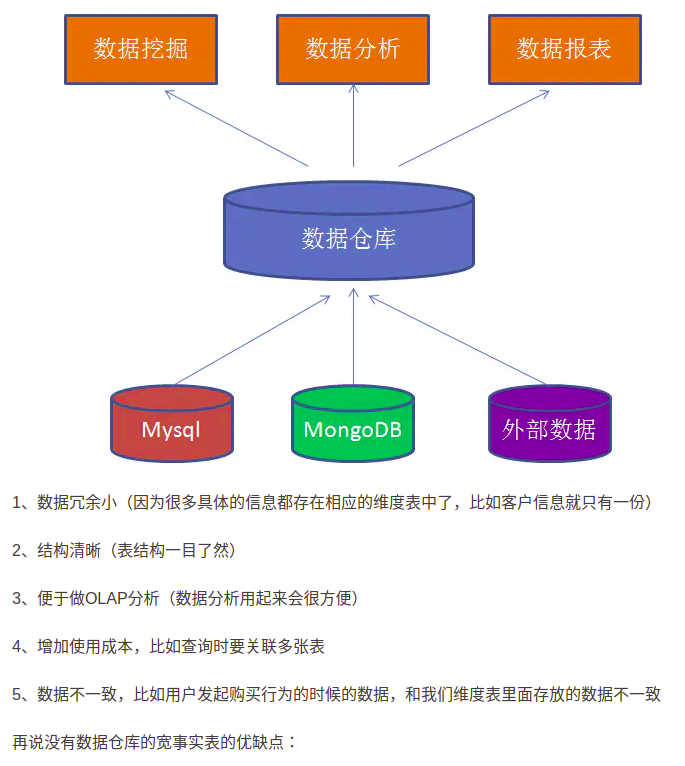
## 事实表和维度表

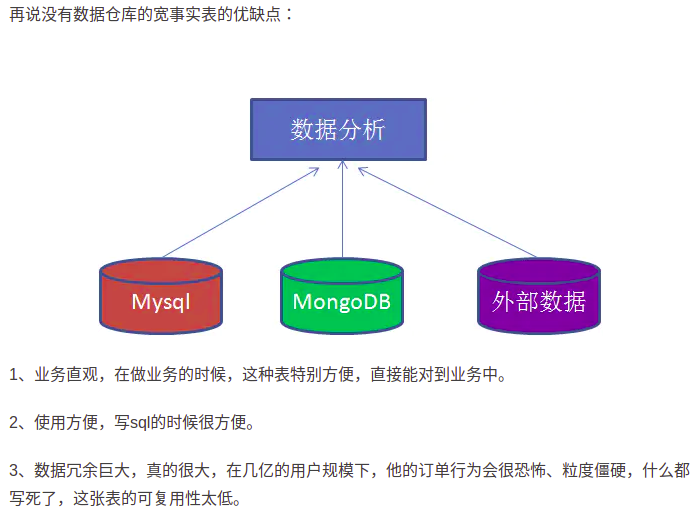
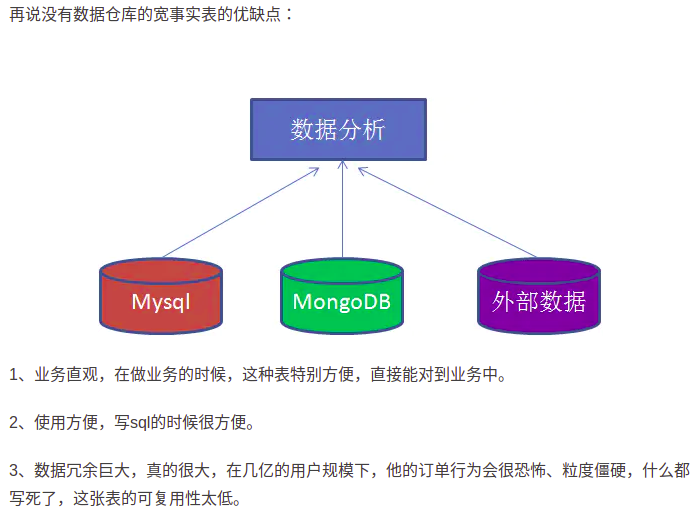
事实表里没有存放实际的内容，他是一堆主键的集合，这些ID分别能对应到维度表中的一条记录。

维度表都包含单一的主键列。维度表的主键可以作为与之关联的任何事实表的外键，当然，维度表行的描述环境应与事实表行完全对应。维度表通常比较宽，是扁平型非规范表，包含大量的低粒度的文本属性。

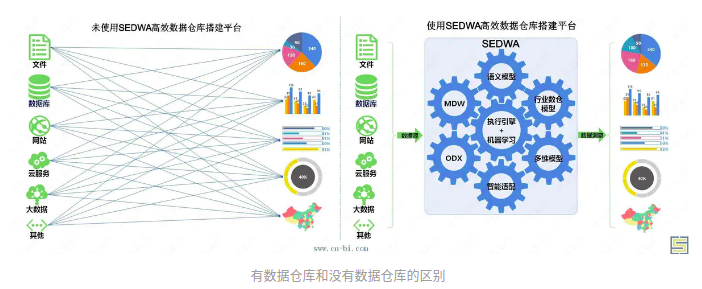


## 数据仓库的优缺点





## 数据仓库对数据分析的作用



数据分析的理解大家应该都是比较熟悉的，数据分析的流程有很多，首先需要进行对业务的理解，然后就是对数据的理解，挖掘数据，数据处理，数据分析，数据展现，这些步骤就能够给大家带来一个不错的数据分析结果。

但是数据分析中的工作最重要的就是数据处理工作，由于数据分析对数据质量、格式的要求比较高，对数据的理解也必须非常深刻，使得数据契合业务需求也要一定的过程，根据我做数据分析的经验，在整个数据分析流程中，用于数据处理的时间往往要占据70%以上。 所以，如何高效、快速地进行数据理解和处理，往往决定了数据分析项目的进度和质量。而数据仓库具有集成、稳定、高质量等特点，基于数据仓库为数据分析提供数据，往往能够更加保证数据质量和数据完整性。

## 数仓分层

**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。

**数据仓库标准上可以分为四层**：ODS（临时存储层）、PDW（数据仓库层）、MID（数据集市层）、APP（应用层）

**ODS层：**

为临时存储层，是接口数据的临时存储区域，为后一步的数据处理做准备。一般来说ODS层的数据和源系统的数据是同构的，主要目的是简化后续数据加工处理的工作。从数据粒度上来说ODS层的数据粒度是最细的。ODS层的表通常包括两类，一个用于存储当前需要加载的数据，一个用于存储处理完后的历史数据。历史数据一般保存3-6个月后需要清除，以节省空间。但不同的项目要区别对待，如果源系统的数据量不大，可以保留更长的时间，甚至全量保存；

**PDW层：**

为数据仓库层，PDW层的数据应该是一致的、准确的、干净的数据，即对源系统数据进行了清洗（去除了杂质）后的数据。这一层的数据一般是遵循数据库第三范式的，其数据粒度通常和ODS的粒度相同。在PDW层会保存BI系统中所有的历史数据，例如保存10年的数据。

**MID层：**

为数据集市层，这层数据是面向主题来组织数据的，通常是星形或雪花结构的数据。从数据粒度来说，这层的数据是轻度汇总级的数据，已经不存在明细数据了。从数据的时间跨度来说，通常是PDW层的一部分，主要的目的是为了满足用户分析的需求，而从分析的角度来说，用户通常只需要分析近几年（如近三年的数据）的即可。从数据的广度来说，仍然覆盖了所有业务数据。

**APP层：**

为应用层，这层数据是完全为了满足具体的分析需求而构建的数据，也是星形或雪花结构的数据。从数据粒度来说是高度汇总的数据。从数据的广度来说，则并不一定会覆盖所有业务数据，而是MID层数据的一个真子集，从某种意义上来说是MID层数据的一个重复。从极端情况来说，可以为每一张报表在APP层构建一个模型来支持，达到以空间换时间的目的数据仓库的标准分层只是一个建议性质的标准，实际实施时需要根据实际情况确定数据仓库的分层，不同类型的数据也可能采取不同的分层方法。

**---【补充，也有分三层的】**

**数据缓存层：**

用于存放接口方提供的原始数据的数据库层，此层的表结构与源数据保持基本一致，数据存放时间根据数据量大小和项目情况而定，如果数据量较大，可以只存近期数据，将历史数据进行备份。此层的目的在于数据的中转和备份。

**核心数据层：**

此层的数据在数据缓存层的基础上做了一定程度的整合，称之为数据集市，存储上仍是关系模型。此层的目的在于进行必要的数据整合为下一步多维模型做准备。

**分析应用层：**

此层的数据为根据业务分析需要构造的多维模型数据。数据可以直接用于分析展现。

说明：数据层次的划分可以根据实际项目需要进行裁剪，如果业务相对简单和独立，可以将核心数据层与分析应用层进行合并。另外，分析应用的数据可以来自多维模型的数据，也可以来自关系模型数据甚至原始数据。

# 数据倾斜

## 原因：



## 表现：

任务进度长时间维持在99%（或100%），查看任务监控页面，发现只有少量（1个或几个）reduce子任务未完成。因为其处理的数据量和其他reduce差异过大。

单一reduce的记录数与平均记录数差异过大，通常可能达到3倍甚至更多。 最长时长远大于平均时长。

## 解决方案：

### 1、参数调节：

****hive.map.aggr=true****

Map 端部分聚合，相当于Combiner

****hive.groupby.skewindata=true****

有数据倾斜的时候进行负载均衡，当选项设定为 true，生成的查询计划会有两个 MR Job。第一个 MR Job 中，Map 的输出结果集合会随机分布到 Reduce 中，每个 Reduce 做部分聚合操作，并输出结果，这样处理的结果是相同的 Group By Key 有可能被分发到不同的 Reduce 中，从而达到负载均衡的目的；第二个 MR Job 再根据预处理的数据结果按照 Group By Key 分布到 Reduce 中（这个过程可以保证相同的 Group By Key 被分布到同一个 Reduce 中），最后完成最终的聚合操作

### sql语句调节

****如何Join：****

关于驱动表的选取，选用join key分布最均匀的表作为驱动表

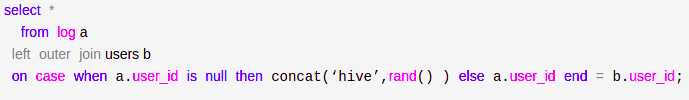
做好列裁剪和filter操作，以达到两表做join的时候，数据量相对变小的效果。

****大小表Join：****

使用map join让小的维度表（1000条以下的记录条数） 先进内存。在map端完成reduce.

****大表Join大表：****

把空值的key变成一个字符串加上随机数，把倾斜的数据分到不同的reduce上，由于null值关联不上，处理后并不影响最终结果。



****count distinct大量相同特殊值****

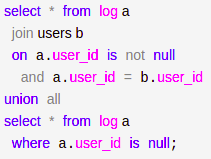
count distinct时，将值为空的情况单独处理，如果是计算count distinct，可以不用处理，直接过滤，在最后结果中加1。如果还有其他计算，需要进行group by，可以先将值为空的记录单独处理，再和其他计算结果进行union。

****group by维度过小：****

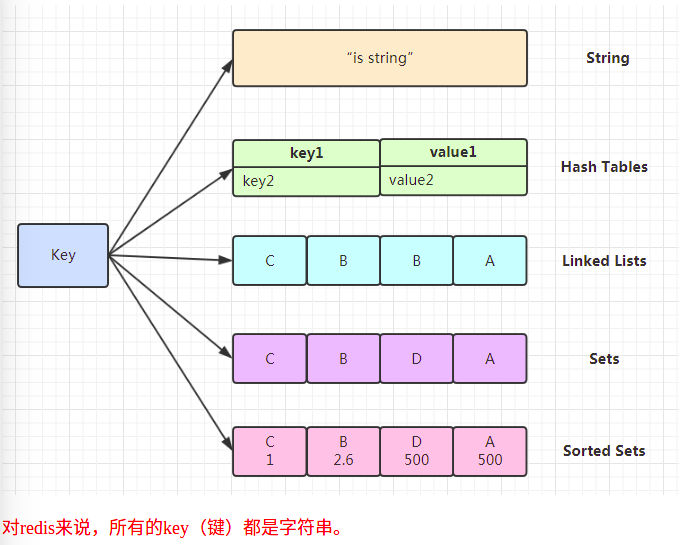
采用sum() group by的方式来替换count(distinct)完成计算。

****特殊情况特殊处理：****

在业务逻辑优化效果的不大情况下，有些时候是可以将倾斜的数据单独拿出来处理。最后union回去。



# redis的五种数据结构

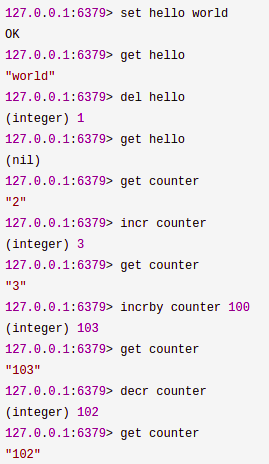


## 1.String 字符串类型

是redis中最基本的数据类型，一个key对应一个value。

String类型是二进制安全的，意思是 redis 的 string 可以包含任何数据。如数字，字符串，jpg图片或者序列化的对象。

使用：get 、 set 、 del 、 incr、 decr 等



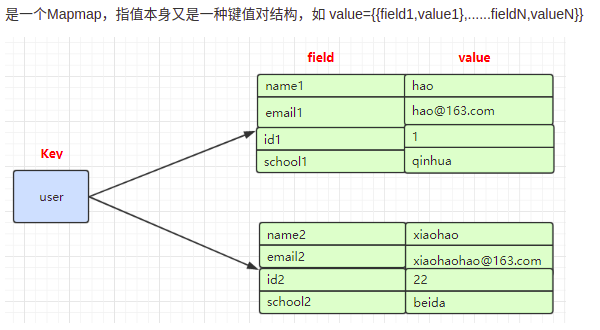
实战场景：

1.缓存： 经典使用场景，把常用信息，字符串，图片或者视频等信息放到redis中，redis作为缓存层，mysql做持久化层，降低mysql的读写压力。

2.计数器：redis是单线程模型，一个命令执行完才会执行下一个，同时数据可以一步落地到其他的数据源。

3.session：常见方案spring session + redis实现session共享

## 2.Hash （哈希）



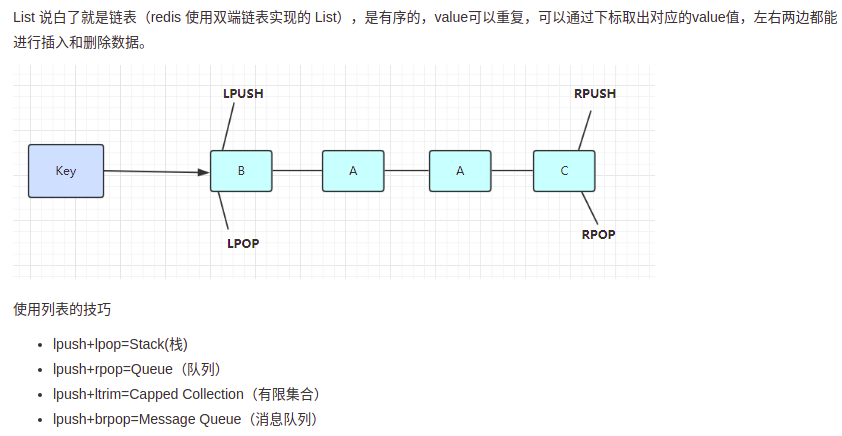
使用：所有hash的命令都是  h   开头的     hget  、hset 、  hdel 等

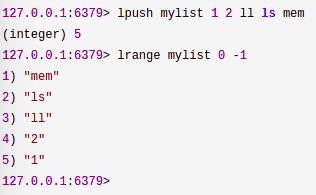


实战场景：

1. 缓存： 能直观，相比string更节省空间

## 3.双端链表

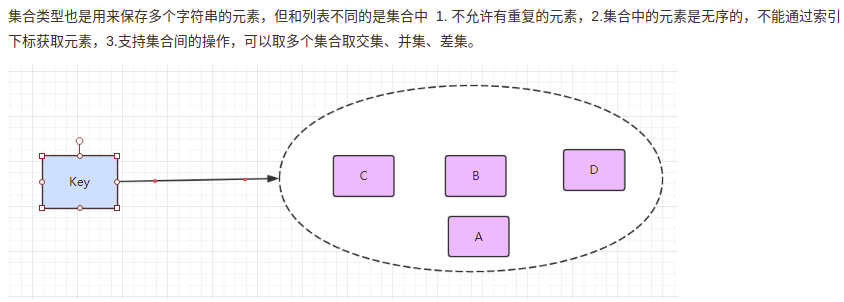




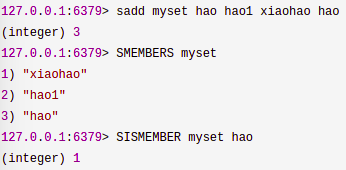
实战场景：

1.timeline：例如微博的时间轴，有人发布微博，用lpush加入时间轴，展示新的列表信息。

## 4.Set 集合



使用：命令都是以s开头的  sset 、srem、scard、smembers、sismember



实战场景;

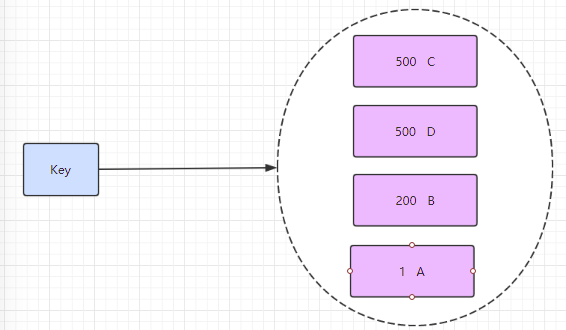
1.标签（tag）,给用户添加标签，或者用户给消息添加标签，这样有同一标签或者类似标签的可以给推荐关注的事或者关注的人。

2.点赞，或点踩，收藏等，可以放到set中实现

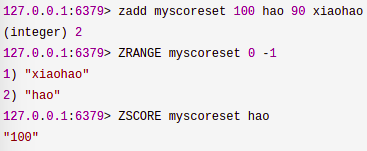
## 5.zset 有序集合

有序集合和集合有着必然的联系，保留了集合不能有重复成员的特性，区别是，有序集合中的元素是可以排序的，它给每个元素设置一个分数，作为排序的依据。

（有序集合中的元素不可以重复，但是score 分数 可以重复，就和一个班里的同学学号不能重复，但考试成绩可以相同）



 有序集合的命令都是 以  z  开头    zadd 、 zrange、 zscore



实战场景：

1.排行榜：有序集合经典使用场景。例如小说视频等网站需要对用户上传的小说视频做排行榜，榜单可以按照用户关注数，更新时间，字数等打分，做排行。

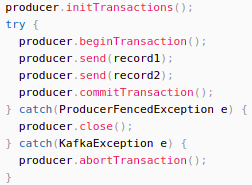
# 9、Kafka的精确一次消费

#### **幂等性：每个分区中精确一次且有序（Idempotence: Exactly-once in order semantics per partition）**

       一个幂等性的操作就是一种被执行多次造成的影响和只执行一次造成的影响一样的操作。现在生产者发送的操作是幂等的了。如果出现导致生产者重试的错误，同样的消息，仍由同样的生产者发送多次，将只被写到kafka broker的日志中一次。对于单个分区，幂等生产者不会因为生产者或broker故障而发送多条重复消息。想要开启这个特性，获得每个分区内的精确一次语义，也就是说没有重复，没有丢失，并且有序的语义，只需要设置producer配置中的"enable.idempotence=true"。  
       这个特性是怎么实现的呢？在底层，它和TCP的工作原理有点像，每一批发送到Kafka的消息都将包含一个序列号，broker将使用这个序列号来删除重复的发送。和只能在瞬态内存中的连接中保证不重复的TCP不同，这个序列号被持久化到副本日志，所以，即使分区的leader挂了，其他的broker接管了leader，新leader仍可以判断重新发送的是否重复了。这种机制的开销非常低：每批消息只有几个额外的字段。你将在这篇文章的后面看到，这种特性比非幂等的生产者只增加了可忽略的性能开销。

#### **事务：跨分区原子写入（Transactions: Atomic writes across multiple partitions）**

       Kafka现在通过新的事务API支持跨分区原子写入。这将允许一个生产者发送一批到不同分区的消息，这些消息要么全部对任何一个消费者可见，要么对任何一个消费者都不可见。这个特性也允许你在一个事务中处理消费数据和提交消费偏移量，从而实现端到端的精确一次语义。下面是的代码片段演示了事务API的使用：



上面的代码片段演示了你可以如何使用新生产者API来原子性地发送消息到topic的多个partition。值得注意的是，一个Kafka topic的分区中的消息，可以有些是在事务中，有些不在事务中。  
       因此在消费者方面，你有两种选择来读取事务性消息，通过隔离等级“isolation.level”消费者配置表示：

read\_commited：除了读取不属于事务的消息之外，还可以读取事务提交后的消息。

read\_uncommited：按照偏移位置读取所有消息，而不用等事务提交。这个选项类似Kafka消费者的当前语义。

       为了使用事务，需要配置消费者使用正确的隔离等级，使用新版生产者，并且将生产者的“transactional.id”配置项设置为某个唯一ID。 需要此唯一ID来提供跨越应用程序重新启动的事务状态的连续性。

# 10、spark和mr的不同：

1、**spark把运算的中间数据存放在内存，迭代计算效率更高**；mapreduce的中间结果需要落地，需要保存到磁盘，这样必然会有磁盘io操做，影响性能。

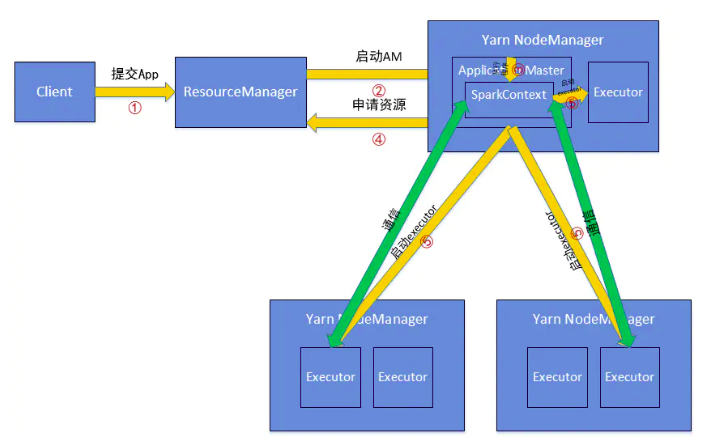
2、**spark容错性高，它通过弹性分布式数据集RDD来实现高效容错**，RDD是一组分布式的存储在节点内存中的只读性质的数据集，这些集合是弹性的，某一部分丢失或者出错，可以通过整个数据集的计算流程的血缘关系来实现重建；mapreduce的话容错可能只能重新计算了，成本较高。

3、**spark更加通用，spark提供了transformation和action这两大类的多个功能api**，另外还有流式处理sparkstreaming模块、图计算GraphX等等；mapreduce只提供了map和reduce两种操作，流计算以及其他模块的支持比较缺乏。

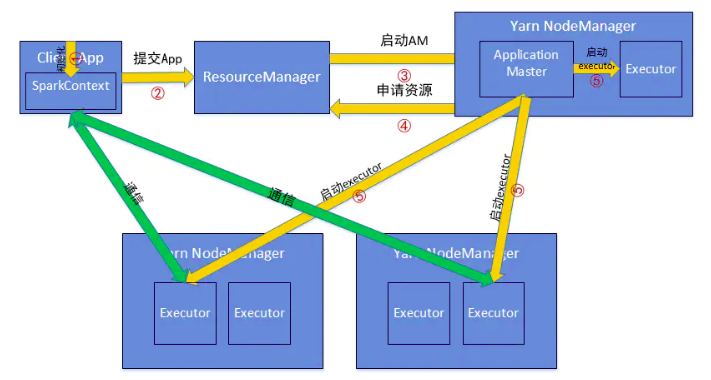
4、**spark框架和生态更为复杂，**首先有RDD、血缘lineage、执行时的有向无环图DAG、stage划分等等，很多时候spark作业都需要根据不同业务场景的需要进行调优已达到性能要求；mapreduce框架及其生态相对较为简单，对性能的要求也相对较弱，但是运行较为稳定，适合长期后台运行。

# 11、spark 在 client 与在集群运行的区别

**cluster**



**Client**



（1）SparkContext初始化不同，这也导致了Driver所在位置的不同，YarnCluster的Driver是在集群的某一台NM上，但是Yarn-Client就是在RM所在机器上；

（2）而Driver会和Executors进行通信，这也导致了Yarn\_cluster在提交App之后可以关闭Client，而Yarn-Client不可以；

（3）最后再来说应用场景，Yarn-Cluster适合生产环境，Yarn-Client适合交互和调试。

# 12、spark比hadoop快的原因：

Spark SQL比Hadoop Hive快，是有一定条件的，而且不是Spark SQL的引擎比Hive的引擎快，相反，Hive的HQL引擎还比Spark SQL的引擎更快。

其实，关键还是在于Spark 本身快。

Spark为什么快？

**1、消除了冗余的HDFS读写**

Hadoop每次shuffle操作后，必须写到磁盘，而Spark在shuffle后不一定落盘，可以cache到内存中，以便迭代时使用。如果操作复杂，

很多的shufle操作，那么Hadoop的读写IO时间会大大增加。

**2、消除了冗余的MapReduce阶段**

Hadoop的shuffle操作一定连着完整的MapReduce操作，冗余繁琐。而Spark基于RDD提供了丰富的算子操作，且reduce操作产生shuffle数据，可以缓存在内存中。

**3、JVM的优化**

Hadoop每次MapReduce操作，启动一个Task便会启动一次JVM，基于进程的操作。

而Spark每次MapReduce操作是基于线程的，只在启动Executor是启动一次JVM，内存的Task操作是在线程复用的。

每次启动JVM的时间可能就需要几秒甚至十几秒，那么当Task多了，这个时间Hadoop不知道比Spark慢了多少。

**4、Hadoop也可能比Spark快**

考虑一种极端查询：Select month\_id,sum(sales) from T group by month\_id;

这个查询只有一次shuffle操作，此时，也许Hive HQL的运行时间也许比Spark还快。

**结论**：Spark快不是绝对的，但是绝大多数，Spark都比Hadoop计算要快。这主要得益于其对mapreduce操作的优化以及对JVM使用的优化。

# Spark SQL基本原理

## **1.1 什么是spark SQL**

​ Spark SQL是Spark用来处理结构化数据的一个模块，它提供了一个编程抽象叫做DataFrame并且作为分布式SQL查询引擎的作用。类似于hive的作用。

## **1.2 spark SQL的特点**

1、容易集成：安装Spark的时候，已经集成好了。不需要单独安装。  
2、统一的数据访问方式：JDBC、JSON、Hive、parquet文件（一种列式存储文件，是SparkSQL默认的数据源，hive中也支持）  
3、完全兼容Hive。可以将Hive中的数据，直接读取到Spark SQL中处理。  
一般在生产中，基本都是使用hive做数据仓库存储数据，然后用spark从hive读取数据进行处理。  
4、支持标准的数据连接：JDBC、ODBC  
5、计算效率比基于mr的hive高，而且hive2.x版本中，hive建议使用spark作为执行引擎

## **2.1 DataFrame和DataSet基本概念**

### **2.1.1 DataFrame**

DataFrame是组织成命名列的数据集。它在概念上等同于关系数据库中的表，里面有表的结构以及数据，但在底层具有更丰富的优化。DataFrames可以从各种来源构建，  
例如：  
结构化数据文件  
hive中的表  
外部数据库或现有RDDs  
DataFrame API支持的语言有Scala，Java，Python和R。

​ 比起RDD，DataFrame多了数据的结构信息，即schema。RDD是分布式的 Java对象的集合。DataFrame是分布式的Row对象的集合。DataFrame除了提供了比RDD更丰富的算子以外，更重要的特点是提升执行效率、减少数据读取以及执行计划的优化。

### **2.1.2 DataSet**

Dataset是一个分布式的数据收集器。这是在Spark1.6之后新加的一个接口，兼顾了RDD的优点（强类型，可以使用功能强大的lambda）以及Spark SQL的执行器高效性的优点。所以可以把DataFrames看成是一种特殊的Datasets，即：Dataset(Row)

## **2.2 创建DataFrame的方式**

### **2.2.1 SparkSession对象**

​ Apache Spark 2.0引入了SparkSession，其为用户提供了一个统一的切入点来使用Spark的各项功能，并且允许用户通过它调用DataFrame和Dataset相关API来编写Spark程序。最重要的是，它减少了用户需要了解的一些概念，使得我们可以很容易地与Spark交互。  
​在2.0版本之前，与Spark交互之前必须先创建SparkConf和SparkContext。然而在Spark 2.0中，我们可以通过SparkSession来实现同样的功能，而不需要显式地创建SparkConf, SparkContext 以及 SQLContext，因为这些对象已经封装在SparkSession中。  
​ 要注意一点，在我用的这个spark版本中，直接使用new SQLContext() 来创建SQLContext对象，会显示该方式已经被弃用了（IDEA中会显示已弃用），建议使用SparkSession来获取SQLContext对象。

### **2.2.2 通过case class样本类**

这种方式在scala中比较常用，因为case class是scala的特色

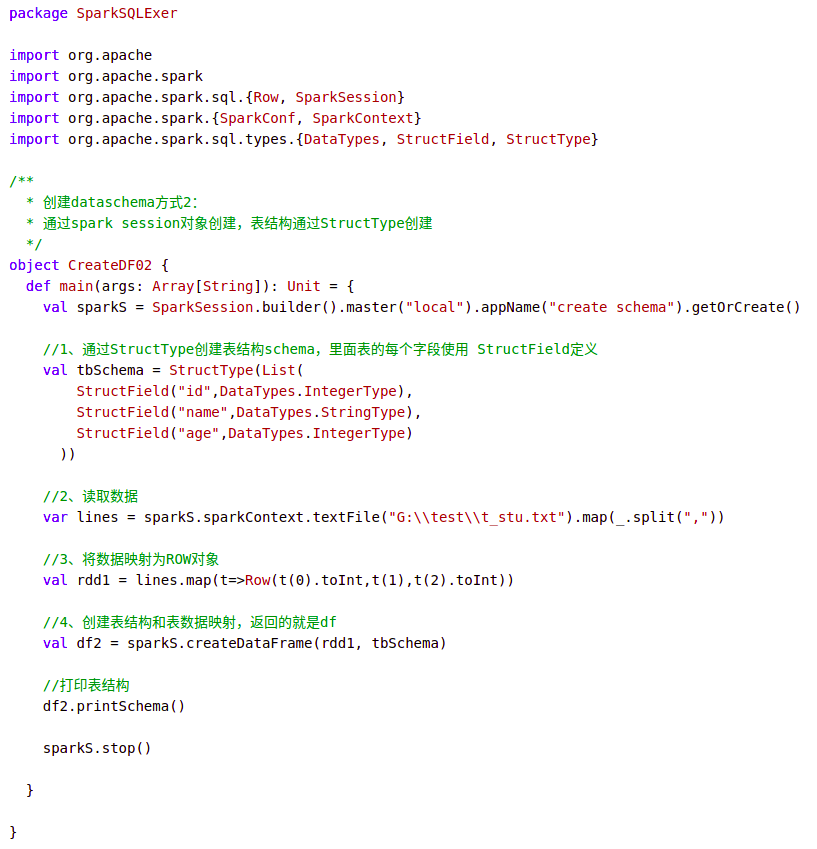


总结步骤为：

* 定义case class，用来表结构
* 创建sparkSession对象，用来读取数据
* 将rdd中的数据和case class映射
* 调用 toDF 函数将rdd转为 DataFrame

### **2.2.3 通过StructType类**

这种方式java比较常用



### **2.2.4 使用json等有表格式的文件类型**



## **2.3 操作DataFrame**

### **2.3.1 DSL语句**

DSL语句其实就是将sql语句的一些操作转为类似函数的方式去调用，比如：

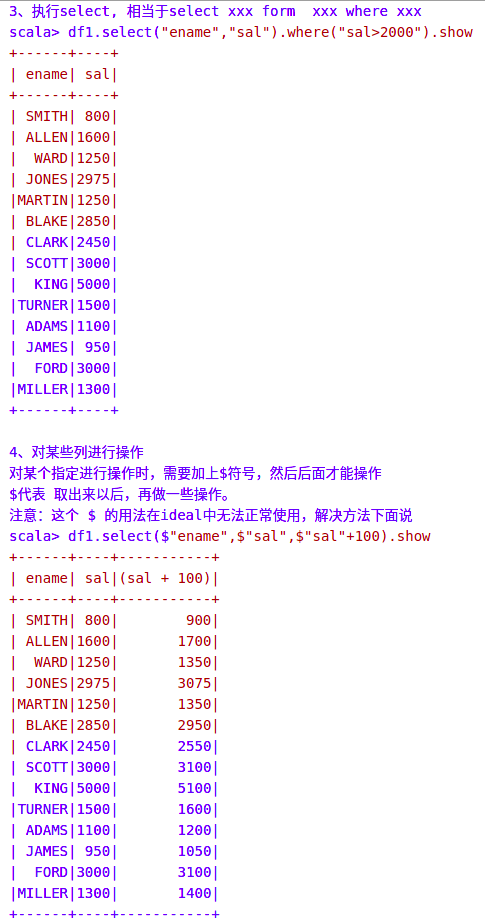
df1.select("name").show

例子：

为了方便，直接在spark-shell里操作了，

spark-shell --master spark://bigdata121:7077

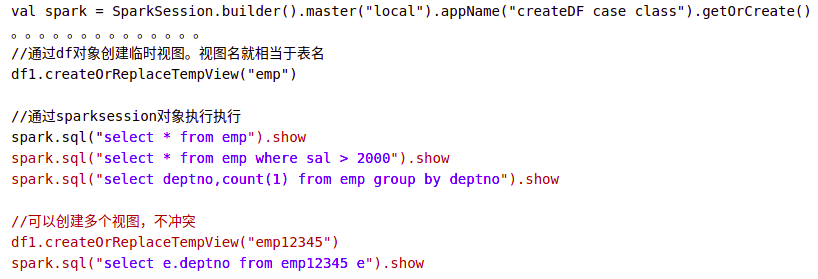




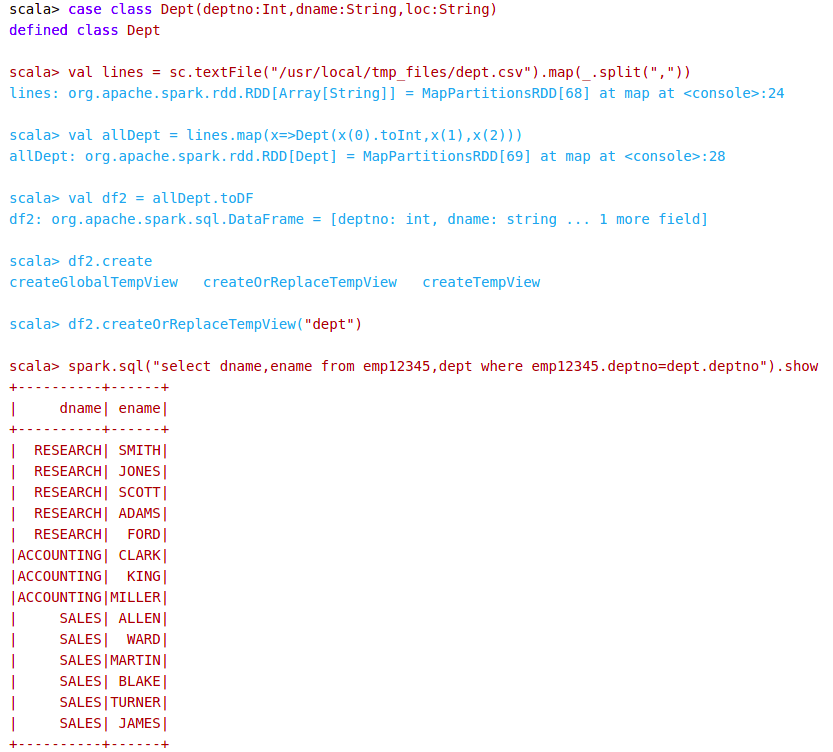


### **2.3.2 sql语句**

df对象不能直接执行sql。需要生成一个视图，再执行SQL。  
需要指定创建的视图名称，后面视图名称就相当于表名。  
视图后面还会细说，这里先有个概念  
例子：



### **2.3.3 多表查询**



## **2.4 创建DataSet**

### **2.4.1 通过case class**

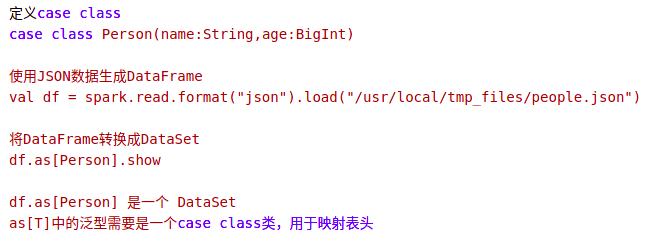
和DataFrame类似，只是把 toDF改为调用 toDS 方法



### **2.4.2 通过序列Seq类对象**



### **2.4.3 使用json格式文件**

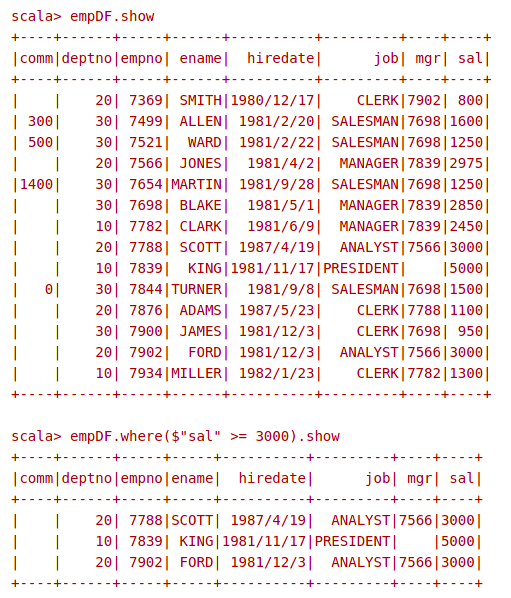


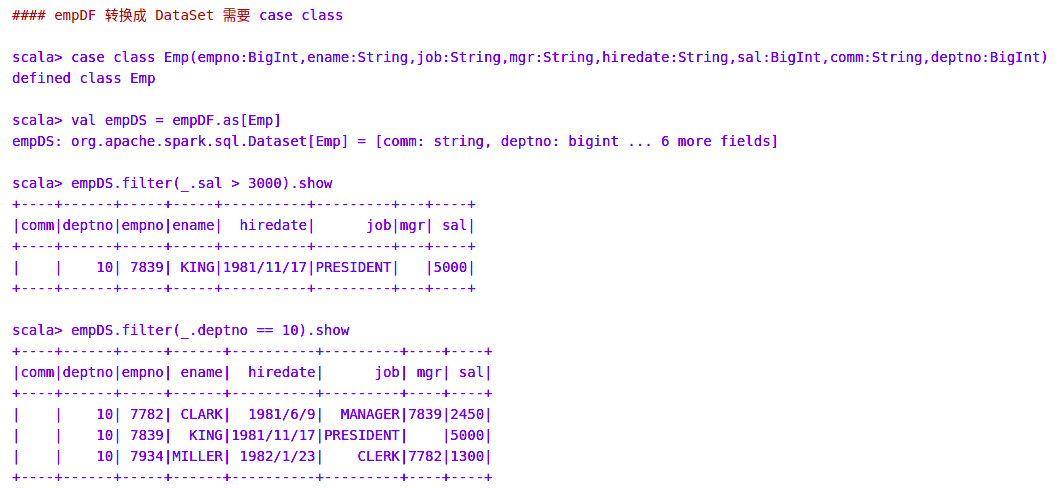
## **2.5 操作DataSet**

DataSet支持的算子其实就是rdd和DataFrame算子的结合。

使用emp.json 生成DataFrame

val empDF = spark.read.json("/usr/local/tmp\_files/emp.json")





**多表查询：**



## **2.6 视图view**



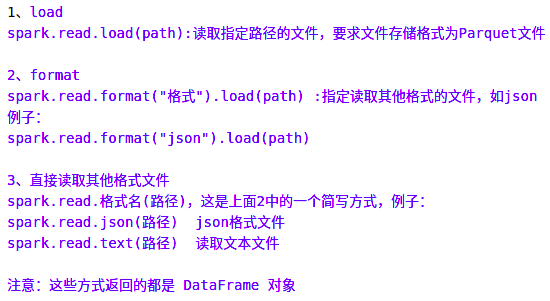
## **2.7 数据源**

通过SparkSession对象可以读取不同格式的数据源：

val spark = SparkSession.builder().master("local").appName("createDF case class").getOrCreate()

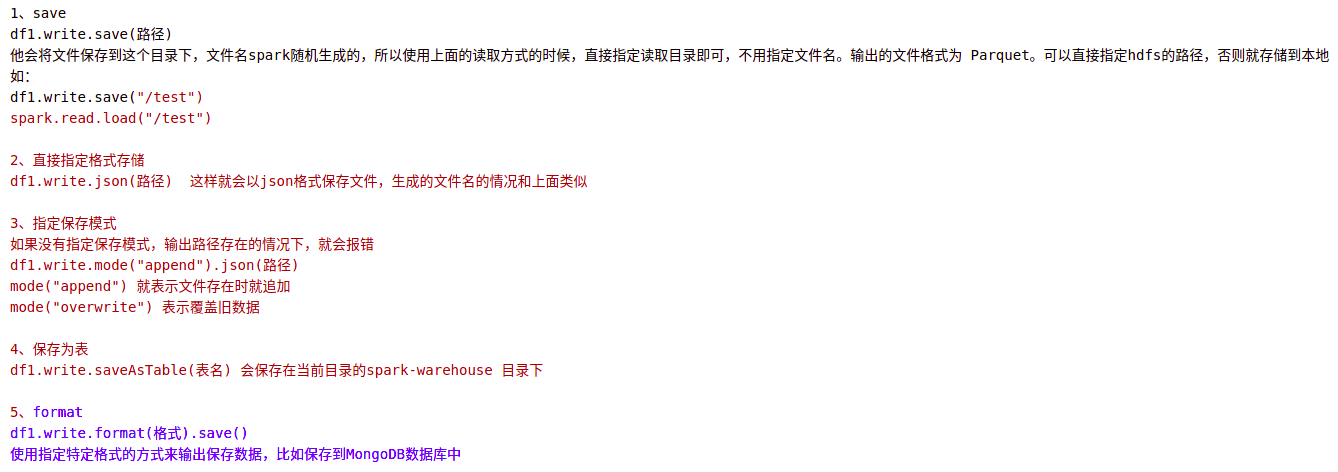
下面都用上面的spark代称SparkSession。

### **2.7.1 SparkSession读取数据的方式**



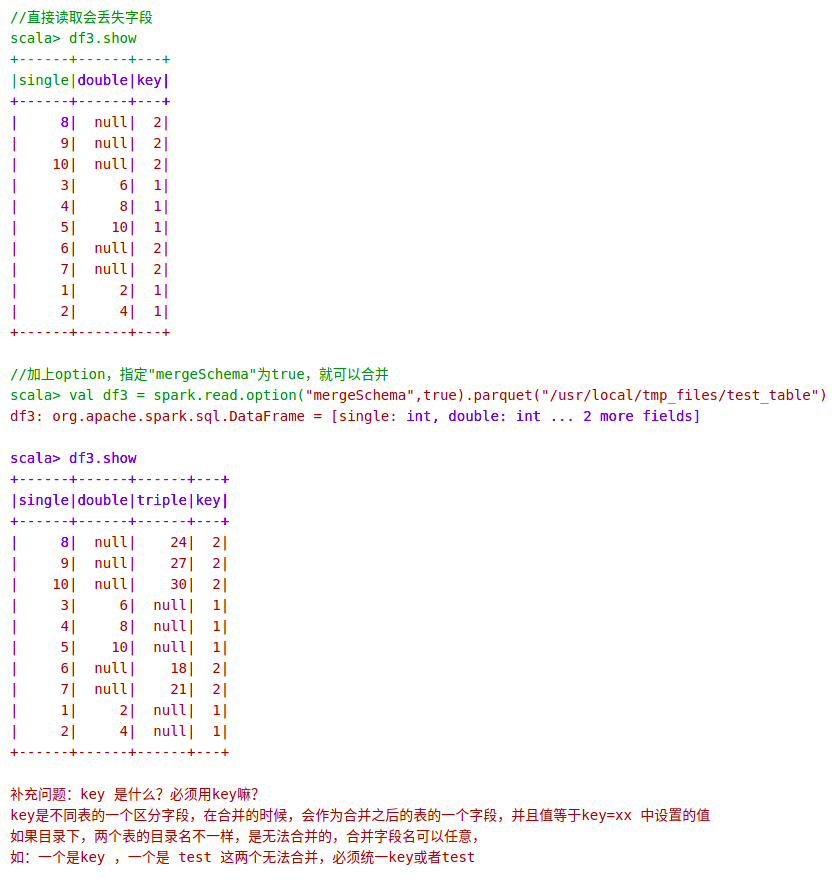
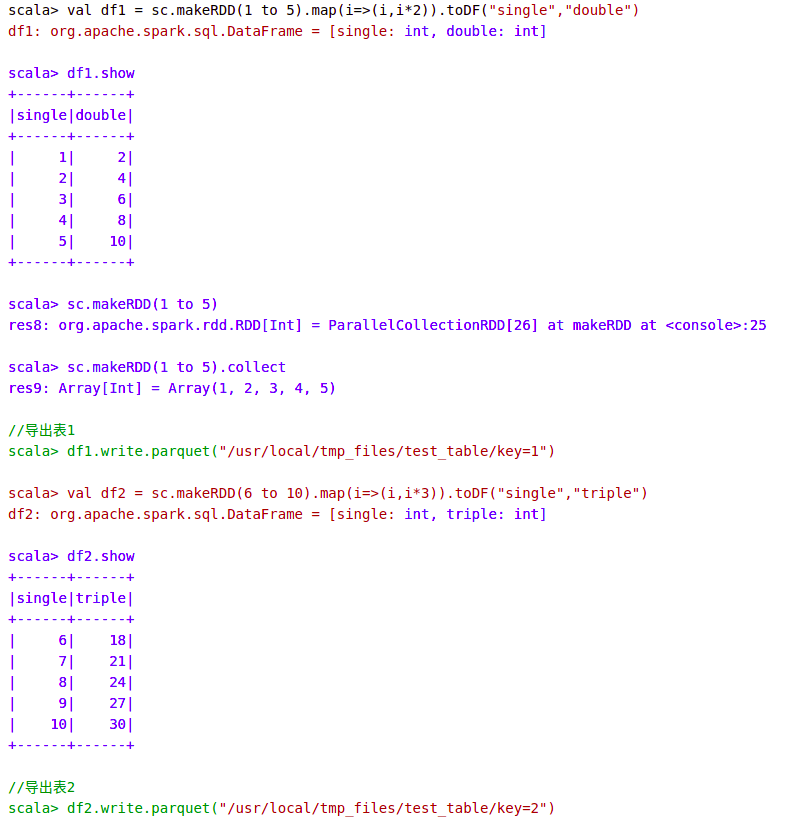
### **2.7.2 SparkSession保存数据的方式**

可以将DataFrame 对象写入到指定格式的文件中，假设有个DataFrame 对象为df1.



### **2.7.3 Parquet格式**

​ 这种一种列式存储格式，具体原理可以看看之前hive的文章。这种格式是默认的存储格式，使用load和save时默认的格式，操作方式很像前面说的，这里不重复。这里要讲的是Parquet的一个特殊的功能，支持schema（表结构）的合并。例子：



### **2.7.5 JDBC 连接**

df对象支持通过jdbc连接数据库，写入数据到数据库，或者从数据库读取数据。  
例子：  
1、通过jdbc 从mysql读取数据：

使用 format(xx).option()的方式指定连接数据库的一些参数，比如用户名密码，使用的连接驱动等



2、jdbc写入数据到mysql

和读取类似，只不过换成了write操作



### **2.7.6 hive**

**1、通过jdbc连接hive**  
方式和普通jdbc类似，例如：



## **2.8 小案例--读取hive数据分析结果写入mysql**

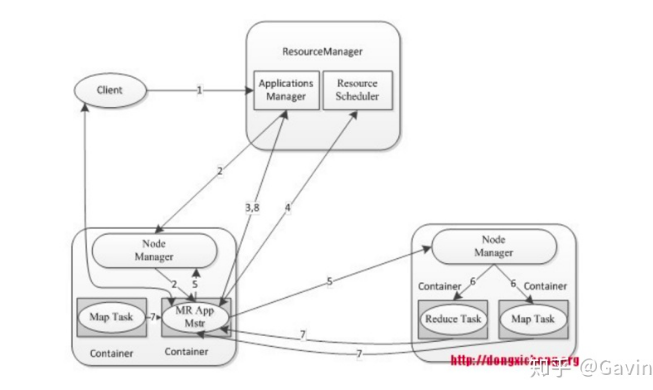


# 14、**简单说一下hadoop和spark的shuffle相同和差异？**

1）从 high-level 的角度来看，两者并没有大的差别。 都是将 mapper（Spark 里是 ShuffleMapTask）的输出进行 partition，不同的 partition 送到不同的 reducer（Spark 里 reducer 可能是下一个 stage 里的 ShuffleMapTask，也可能是 ResultTask）。Reducer 以内存作缓冲区，边 shuffle 边 aggregate 数据，等到数据 aggregate 好以后进行 reduce() （Spark 里可能是后续的一系列操作）。

2）从 low-level 的角度来看，两者差别不小。 Hadoop MapReduce 是 sort-based，进入 combine() 和 reduce() 的 records 必须先 sort。这样的好处在于 combine/reduce() 可以处理大规模的数据，因为其输入数据可以通过外排得到（mapper 对每段数据先做排序，reducer 的 shuffle 对排好序的每段数据做归并）。目前的 Spark 默认选择的是 hash-based，通常使用 HashMap 来对 shuffle 来的数据进行 aggregate，不会对数据进行提前排序。如果用户需要经过排序的数据，那么需要自己调用类似 sortByKey() 的操作；如果你是Spark 1.1的用户，可以将spark.shuffle.manager设置为sort，则会对数据进行排序。在Spark 1.2中，sort将作为默认的Shuffle实现。

# 15、spark cluster模式执行流程



1）客户端client向ResouceManager提交Application，ResouceManager接受Application

并根据集群资源状况选取一个node来启动Application的任务调度器driver（ApplicationMaster）

2）ResouceManager找到那个node，命令其该node上的nodeManager来启动一个新的JVM进程运行程序的driver（ApplicationMaster）部分，driver（ApplicationMaster）启动时会首先向ResourceManager注册，说明由自己来负责当前程序的运行。

3）driver（ApplicationMaster）开始下载相关jar包等各种资源，基于下载的jar等信息决定向ResourceManager申请具体的资源内容。

4）ResouceManager接受到driver（ApplicationMaster）提出的申请后，会最大化的满足资源分配请求，并发送资源的元数据信息给driver（ApplicationMaster）；

5）driver（ApplicationMaster）收到发过来的资源元数据信息后会根据元数据信息发指令给具体机器上的NodeManager，让其启动具体的container。

6）NodeManager收到driver发来的指令，启动container，container启动后必须向driver（ApplicationMaster）注册。

7）driver（ApplicationMaster）收到container的注册，开始进行任务的调度和计算，直到任务完成。

补充：如果ResourceManager第一次没有能够满足driver（ApplicationMaster）的资源请求，后续发现有空闲的资源，会主动向driver（ApplicationMaster）发送可用资源的元数据信息以提供更多的资源用于当前程序的运行。

# zookeeper中paxos协议大白话

假设有一个社团，其中有团员、议员（决议小组成员）两个角色

团员可以向议员申请提案来修改社团制度

议员坐在一起，拿出自己收到的提案，对每个提案进行投票表决，超过半数通过即可生效

为了秩序，规定每个提案都有编号ID，按顺序自增

每个议员都有一个社团制度笔记本，上面记着所有社团制度，和最近处理的提案编号，初始为0

投票通过的规则：

新提案ID 是否大于 议员本中的ID，是议员举手赞同

如果举手人数大于议员人数的半数，即让新提案生效

例如：

刚开始，每个议员本子上的ID都为0，现在有一个议员拿出一个提案：团费降为100元，这个提案的ID自增为1

每个议员都和自己ID对比，一看 1>0，举手赞同，同时修改自己本中的ID为1

发出提案的议员一看超过半数同意，就宣布：1号提案生效

然后所有议员都修改自己笔记本中的团费为100元

以后任何一个团员咨询任何一个议员："团费是多少？"，议员可以直接打开笔记本查看，并回答：团费为100元

可能会有极端的情况，就是多个议员一起发出了提案，就是并发的情况

例如

刚开始，每个议员本子上的编号都为0，现在有两个议员（A和B）同时发出了提案，那么根据自增规则，这两个提案的编号都为1，但只会有一个被先处理

假设A的提案在B的上面，议员们先处理A提案并通过了，这时，议员们的本子上的ID已经变为了1，接下来处理B的提案，由于它的ID是1，不大于议员本子上的ID，B提案就被拒绝了，B议员需要重新发起提案

上面就是Paxos的基本思路，对照ZooKeeper，对应关系就是：

团员 -client

议员 -server

议员的笔记本 -server中的数据

提案 -变更数据的请求

提案编号 -zxid（ZooKeeper Transaction Id）

提案生效 -执行变更数据的操作

ZooKeeper中还有一个leader的概念，就是把发起提案的权利收紧了，以前是每个议员都可以发起提案，现在有了leader，大家就不要七嘴八舌了，先把提案都交给leader，由leader一个个发起提案

Paxos算法就是通过投票、全局编号机制，使同一时刻只有一个写操作被批准，同时并发的写操作要去争取选票，只有获得过半数选票的写操作才会被批准，所以永远只会有一个写操作得到批准，其他的写操作竞争失败只好再发起一轮投票

**1）一致性保证**

Zookeeper是一种高性能、可扩展的服务。Zookeeper的读写速度非常快，并且读的速度要比写的速度更快。另外，在进行读操作的时候，ZooKeeper依然能够为旧的数据提供服务。这些都是由于ZooKeepe所提供的一致性保证，它具有如下特点：

 顺序一致性

客户端的更新顺序与它们被发送的顺序相一致。

原子性

更新操作要么成功要么失败，没有第三种结果。

单系统镜像

无论客户端连接到哪一个服务器，客户端将看到相同的ZooKeeper视图。

 可靠性

一旦一个更新操作被应用，那么在客户端再次更新它之前，它的值将不会改变。。这个保证将会产生下面两种结果：

1．如果客户端成功地获得了正确的返回代码，那么说明更新已经成功。如果不能够获得返回代码（由于通信错误、超时等等），那么客户端将不知道更新操作是否生效。

2．当从故障恢复的时候，任何客户端能够看到的执行成功的更新操作将不会被回滚。

 实时性

在特定的一段时间内，客户端看到的系统需要被保证是实时的（在十几秒的时间里）。在此时间段内，任何系统的改变将被客户端看到，或者被客户端侦测到。

给予这些一致性保证，ZooKeeper更高级功能的设计与实现将会变得非常容易，例如：leader选举、队列以及可撤销锁等机制的实现。

**2）Leader选举**

ZooKeeper需要在所有的服务（可以理解为服务器）中选举出一个Leader，然后让这个Leader来负责管理集群。此时，集群中的其它服务器则成为此Leader的Follower。并且，当Leader故障的时候，需要ZooKeeper能够快速地在Follower中选举出下一个Leader。这就是ZooKeeper的Leader机制，下面我们将简单介绍在ZooKeeper中，Leader选举（Leader Election）是如何实现的。

此操作实现的核心思想是：首先创建一个EPHEMERAL目录节点，例如“/election”。然后。每一个ZooKeeper服务器在此目录下创建一个SEQUENCE|[EPHEMERAL](/home/vincent/Documents\\x/" \t "/home/vincent/Documents\\x/_blank)类型的节点，例如“/election/n\_”。在SEQUENCE标志下，ZooKeeper将自动地为每一个ZooKeeper服务器分配一个比前一个分配的序号要大的序号。此时创建节点的ZooKeeper服务器中拥有最小序号编号的服务器将成为Leader。

在实际的操作中，还需要保障：当Leader服务器发生故障的时候，系统能够快速地选出下一个ZooKeeper服务器作为Leader。一个简单的解决方案是，让所有的follower监视leader所对应的节点。当Leader发生故障时，Leader所对应的临时节点将会自动地被删除，此操作将会触发所有监视Leader的服务器的watch。这样这些服务器将会收到Leader故障的消息，并进而进行下一次的Leader选举操作。但是，这种操作将会导致“从众效应”的发生，尤其当集群中服务器众多并且带宽延迟比较大的时候，此种情况更为明显。

在Zookeeper中，为了避免从众效应的发生，它是这样来实现的：每一个follower对follower集群中对应的比自己节点序号小一号的节点（也就是所有序号比自己小的节点中的序号最大的节点）设置一个watch。只有当follower所设置的watch被触发的时候，它才进行Leader选举操作，一般情况下它将成为集群中的下一个Leader。很明显，此Leader选举操作的速度是很快的。因为，每一次Leader选举几乎只涉及单个follower的操作。

1. 设计微信群发红包的的数据库表格式:

drop table if exists wc\_groupsend\_rp;  
create external table wc\_groupsend\_rp (  
     wcgroupName string, --群名称  
     imid string, --发红包的设备ID  
     wcid string, --发红包的微信号  
     wcname string, --发红包的微信名  
     rpamount double, --红包金额  
     rpid string, --红包标识， 唯一确定一个红包  
     rpcount int, --红包数量  
     rptype int, --红包类型 比如1拼手气红包,2为普通红包,3为指定人领取红包  
     giverpdt string, --发红包时间   
     setuprpdt string, --创建红包时间 点击红包按钮的时间

paydt string, --支付时间  
) COMMENT '群发红包表'  
PARTITIONED BY (`giverpdt` string)  
row format delimited fields terminated by '\t';

// 因为发红包的时间确定，所以用发红包的时间范围来划分分区

 drop table if exists wc\_groupcash\_rp;  
create external table wc\_groupcash\_rp (   
    rpid string, -- 红包标识  
     imid string, -- 发红包的设备ID  
     wcid string, -- 发红包的微信号  
     wcname string, -- 发红包的微信名   
     wcgroupName string, --群名称  
     cashdt stirng, --红包领取时间 每领取一次更新一条数据   
     cashcount int, --领取人数  
     cashamount double, --领取金额  
     cashwcid string, --领取人的微信  
     cashwcname string, --领取人微信昵称  
     cashsum double, --已领取总金额  
) COMMENT '红包领取表'  
PARTITIONED BY (`rpid` string)  
row format delimited fields terminated by '\t';

//因为领取红包的时间不同，所以用发出红包的pid来做范围分区。