1.1 针对海量数据记录如何去重？说出尽可能多的方式

1. 如果100亿条数据能在内存中放得下，可以选择排序，相同记录会排在一起，只取第一条。
2. 利用HashMap，记录为key设置HashMap，遍历所有数据一遍后，最后HashMap的key就是所有去重后的记录,前提是海量文件存储的Map没有超出内存限制。
3. 如果数据量超出内存，可以采用针对海量数据记录进行文件切割方式，处理成一个个小文件，对每个小文件进行归并排序找出重复数据只保留一条即可。
4. bitmap位图操作，位图优势是存储空间小速度又快，遍历一遍数据存储在位图中，每次读取数据时判断位图中是否存储过该数据，位图中有对应的数据则代表是重复的数据，可以丢弃不写出。

1.2 Spark on Hive 与 Hive on Spark什么区别？

- **Hive on Spark:**

Hive on Spark是在Hive上新增一种计算引擎：Spark，目的是借助Spark内存计算引擎的优势提升查询Hive数据的性能，默认执行HQL转换成MR，性能慢，底层使用Spark引擎效率高。Hive on Spark与Hive on Tez 、Hive on MR(默认)一样，只是底层执行的引擎不一样而已。

- **Spark on Hive:**

没有官方的Spark on Hive说法，属于大家习惯性的称呼，指的是SparkSQL 读写Hive 表特点场景。SparkSQL可以不读取Hive中的数据，也可以读取Hive中的数据，Spark on Hive目的是让SparkSQL可以访问Hive表，Spark on Hive 就是SparkSQL可以访问Hive表，可以基于SparkSQL构建Hive数仓。

Hive on Spark与Spark on Hive异同点：

- **相同点：**

SQL执行层都是使用Spark执行引擎

- **不同点：**

1. 两者SQL解析层不同，Hive on Spark使用Hive compiler，Spark on Hive 使用的是Spark compiler。
2. Spark on Hive 中 SparkSQL作为Spark生态圈中的一员继续发展，不受限与Hive，只是兼容Hive。
3. Hive on Spark 是Hive中的发展计划，该计划将Spark作为Hive底层引擎之一，Hive 支持引擎除了Spark外还有默认的MR、Tez。

1.3 数据仓库建模方式及数据分析模型有哪些？

数据仓库建模方式包含范式建模和维度建模。ER实体关系模型（Entity-Relationship）是数据库设计的理论基础，当前几乎所有的OLTP系统设计都采用ER模型建模的方式,这种建模方式基于三范式。在信息系统中，将事物抽象为“实体”、“属性”、“关系”来表示数据关联和事物描述。维度建模主要源自数据集市，主要面向分析场景。

维度建模以分析决策的需求出发构建模型，构建的数据模型为分析需求服务，因此它重点解决用户如何更快速完成分析需求，同时还有较好的大规模复杂查询的响应性能。它与实体-关系（ER）建模有很大的区别，实体-关系建模是面向应用，遵循第三范式，以消除数据冗余为目标的设计技术。维度建模是面向分析，为了提高查询性能可以增加数据冗余，反规范化的设计技术。

在多维分析的商业智能解决方案中，根据事实表和维度表的关系，又可将常见的模型分为星型模型、雪花型模型、星座模型。

星型模型：当所有的维度表都由连接键连接到事实表时，结构图如星星一样，这种分析模型就是星型模型。

雪花模型：当有一个或多个维表没有直接连接到事实表上，而是通过其他维表连接到事实表上时，其结构图就像雪花连接在一起，这种分析模型就是雪花模型。

星座模型：当多个事实表共用多张维度表时，就构成了星座模型，可以理解成星座模型有很多星型模型组成，星座模型就是星型模型特例。一个企业中星座模型使用情况居多。

星型模型和雪花模型主要区别就是对维度表的拆分，对于雪花模型，维度表的设计更加规范，一般符合三范式设计;而星型模型，一般采用降维的操作，维度表设计不符合三范式设计，反规范化，利用冗余牺牲空间来避免模型过于复杂，提高易用性和分析效率。

星型模型因为数据的冗余所以很多统计查询不需要做外部的连接，因此一般情况下效率比雪花型模型要高。星型结构不用考虑很多正规化的因素，设计与实现都比较简单。

雪花型模型由于去除了冗余，有些统计就需要通过表的联接才能产生，所以效率不一定有星型模型高。正规化也是一种比较复杂的过程，相应的数据库结构设计、数据的ETL、以及后期的维护都要复杂一些。因此在冗余可以接受的前提下，数仓构建实际运用中星型模型使用更多，也更有效率。

1.4 什么是缓慢变化维（SCD）,通过设计怎样解决缓慢变化维的问题？

在构建数据仓库过程中，将业务数据从关系型数据库中抽取到数据仓库过程中，涉及到一些数据的增量抽取，针对事实数据我们会将事实表的增量数据存储到一个新的分区中解决。对于维度数据的变化

包括修改和增加，在数据仓库中应该如何处理，一般这些维度数据是随着时间发生变化，这种维度数据我们称为缓慢变化维（Slowly Changing Dimensions，简称SCD），并且把处理维度表的历史变化信息的问题称为处理缓慢变化维的问题，有时也简称为处理SCD的问题。

例如：存储在关系型数据库中有一张用户维度表person_info:

uid	name	age	address	dt
uid001	张三	18	北京	2022-01-01
uid002	李四	19	上海	2022-01-02

以上维度表数据导入到数据仓库后也会对应有用户维度表。假设“张三”由于某种原因居住地由“北京”变成了“天津”，那么在业务库中直接更新对应的address字段即可。如果在数仓中直接将对应用户维度表全量更新替换，那么在统计用户所在地区消费额时就有可能造成“张三”之前本属于“北京”地区消费的金额都被统计到了后来的“天津”地区，那么就会引起数据归纳和分析有问题。

在业务库中维度数据的变化是非常自然和正常的，例如：用户的手机号、用户所在地、商品不同时期价格和折扣不同等。在业务数据库中，这种变化的数据通过update修改会马上反应到实际业务中去，但是在数据仓库中为了保证数据分析能反应历史变化，只允许数据增加和查询，不允许对数据进行修改和删除，这样在数据仓库中才能反映出对应数据周期变化的历史，那么对于这种变化的维度数据该如何处理？数据仓库中处理缓慢变化维的方法通常分为三种方式：

第一种方式是直接覆盖原值（也称Type 1 SCD）。

在数据仓库中始终保持和业务维度数据一致，可以根据维度表的uid唯一主键来追踪判断维度数据是否变化，一旦发生变化就将旧的业务数据覆盖重写，这种方式处理最容易实现，但是没有保留历史数据，无法分析历史变化信息。第一种方式常用于维度表中变化的字段不影响后续数据统计分析，例如：国家或者地区名称变化。更新后数据仓库中的表结果如下：

uid	name	age	address	dt
uid001	张三	18	天津	2022-01-03
uid002	李四	19	上海	2022-01-02

第二种方式是添加维度行（也称TYPE 2 SCD）。

数据仓库更多是对相对静态的历史数据进行数据的汇总和分析，因此会尽可能的维护来自业务系统中的历史数据，能够真正分析到历史数据变化。第二种方式就是针对维度表一些列变化的数据在数据仓库对应的维度表中增加新行。

uid	name	age	address	dt
uid001	张三	18	北京	2022-01-01
uid002	李四	19	上海	2022-01-02
uid001	张三	18	天津	2022-01-03

增加新的维度行后，在使用对应维度数据时为了分辨出哪条维度数据是在用的，我们还会通过指定起始时间进行标识，如下,valid_from和valid_to表示数据使用的起始和终止时间，如果valid_to是null代表该条数据是最新在用维度数据。

uid	name	age	address	dt	valid_from	valid_to
uid001	张三	18	北京	2022-01-01	2022-01-01	2022-01-03
uid002	李四	19	上海	2022-01-02	2022-01-02	null
uid001	张三	18	天津	2022-01-03	2022-01-03	null

第三种方式是添加属性列（Type 3 SCD）。

实际上第一种和第二种SCD处理方式可以满足大多数需求，那么Type3 SCD目的主要是维护更少的历史记录。针对最初始的维度数据变化后使用Type3 SCD构建维度表如下：

uid	name	age	curt_ads	dt	pre_ads	pre_dt
uid001	张三	18	天津	2022-01-03	北京	2022-01-01
uid002	李四	19	上海	2022-01-02	null	null

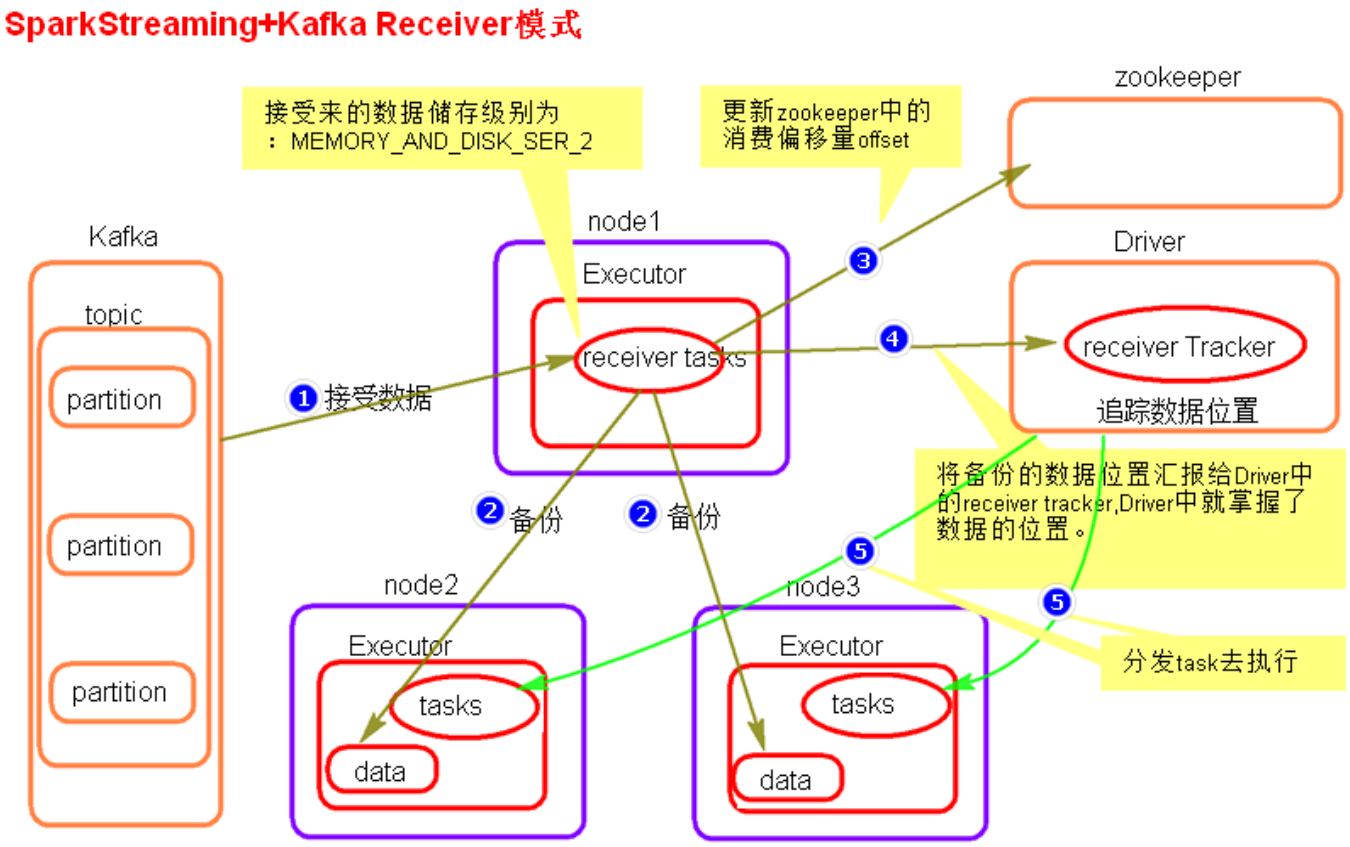
以上处理方式是针对要维护的列增加新的一列“pre_address”,然后每次更新curt_ads和pre_ads即可，这种方式只保存了最近两次的历史记录，但是如果要维护的字段比较多，这种方式就比较麻烦，因为要维护更多的current和previous字段，所以这种方式并没有前两种使用广泛。为了方便后期使用该维度表与事实表进行关联还会构建pre_dt用于判断时间范围决定使用curt_ads和pre_ads中的那个字段。

在实际建模中，我们可以联合使用三种方式，也可以对一个维度表中的不同属性使用不同的方式，这些，都需要根据实际情况来决定，但目的都是一样的，就是能够支持方便的分析历史变化情况。

1.5 SparkStreaming读取Kafka中数据如何保证消费一致性?

1.5.1 Receiver模式

在早期Spark消费Kafka中数据只支持Receiver模式。Receiver模式读取Kafka中数据原理图如下：



Receiver模式中，SparkStreaming使用Receiver接收器模式来接收kafka中的数据，即会将每批次数据都存储在Spark端，默认的存储级别为MEMORY_AND_DISK_SER_2，从Kafka接收过来数据之后，还会将数据备份到其他Executor节点上，当完成备份之后，再将消费者offset数据写往zookeeper中，然后再向Driver汇报数据位置，Driver发送task到数据所在节点处理数据。

这种模式使用zookeeper来保存消费者offset，等到SparkStreaming重启后，从zookeeper中获取offset继续消费。

当Driver挂掉时，同时消费数据的offset已经更新到zookeeper中时，SparkStreaming重启后，接着zookeeper存储的offset继续处理数据，这样就存在丢失数据的问题。

为了解决以上丢失数据的问题，可以开启WAL(write ahead log)预写日志机制，将从kafka中接收来的数据备份完成之后，向指定的checkpoint中也保存一份，这样当SparkStreaming挂掉，重新启动再处理数据时，会处理Checkpoint中最近批次的数据，将消费者offset继续更新保存到zookeeper中。

开启WAL机制，需要设置checpoint,由于一般checkpoint路径都会设置到HDFS中，HDFS本身会有副本，所以这里如果开启WAL机制之后，可以将接收数据的存储级别降级，去掉“_2”级别。

开启WAL机制之后带来了新的问题：

- **数据重复处理问题**

由于开启WAL机制，会处理checkpoint中最近一段时间批次数据，这样会造成重复处理数据问题。所以对于数据需要精准消费的场景，不能使用receiver模式。如果不开启WAL机制Receiver模式有丢失数据的问题，开启WAL机制之后有重复处理数据的问题，对于精准消费数据的场景，只能人为保存offset来保证数据消费的准确性。

- **数据处理延迟加大问题**

数据在节点之间备份完成后再向checkpoint中备份，之后再向Zookeeper汇报数据offset，向Driver汇报数据位置，然后Driver发送task处理数据。这样加大了数据处理过程中的延迟。

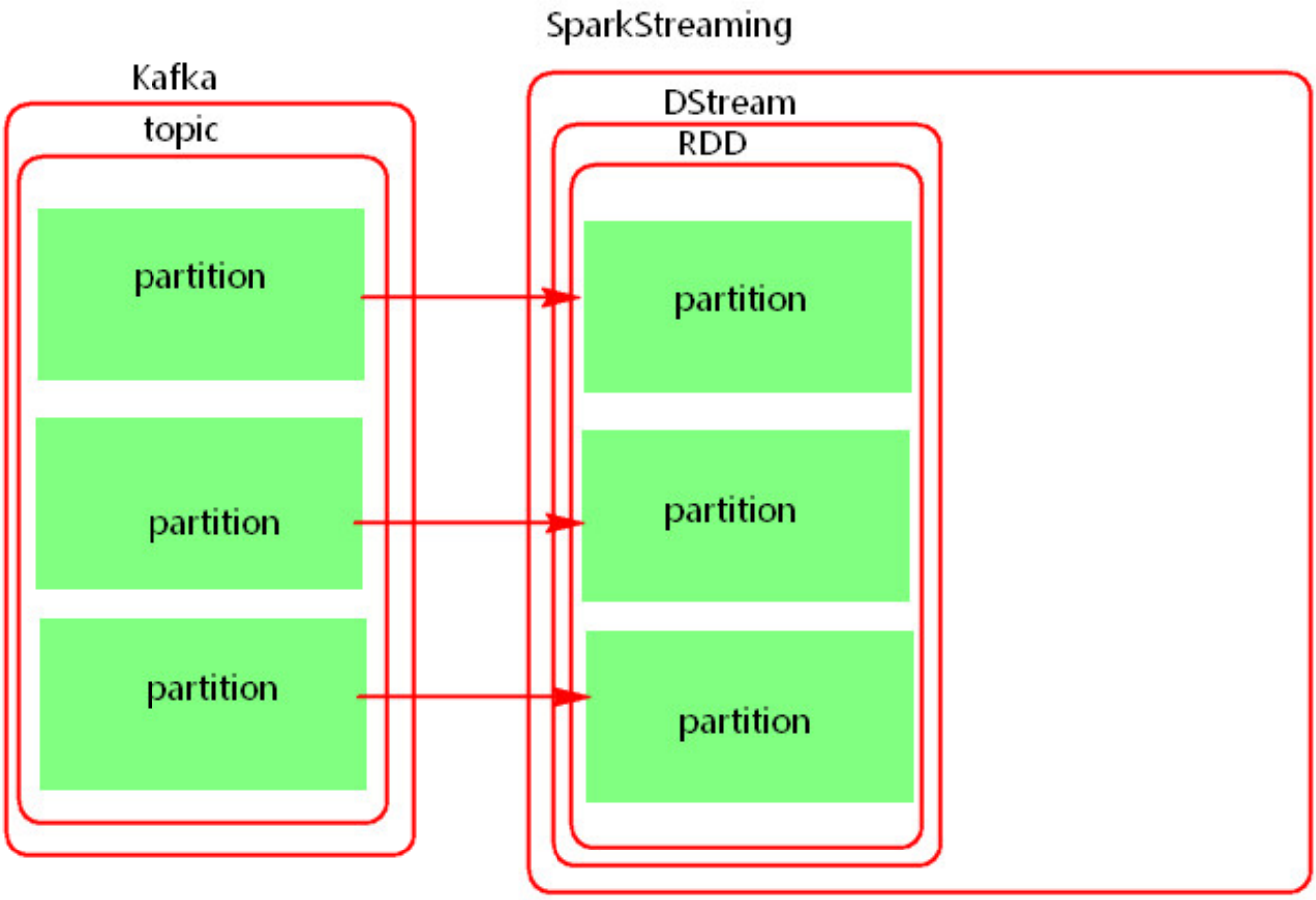
对于精准消费数据的问题，需要我们从每批次中获取offset然后保存到外部的数据库来实现来实现仅一次消费数据。但是Receiver模式底层读取Kafka数据的实现使用的是High Level Consumer Api，这种Api不支持获取每批次消费数据的offset。所以对于精准消费数据的场景不能使用这种模式。

Receiver模式总结

1. Receiver模式采用了Receiver接收器的模式接收数据。会将每批次的数据存储在Executor内存或者磁盘中。
2. Receiver模式有丢失数据问题，开启WAL机制解决，但是带来新的问题。
3. receiver模式依赖zookeeper管理消费者offset。
4. SparkStreaming读取Kafka数据，相当于Kafka的消费者，底层读取Kafka采用了“High Level Consumer API”实现，这种api没有提供操作每批次数据offset的接口，所以对于精准消费数据的场景想要人为控制offset是不可能的。

1.5.2 Direct模式

在Spark1.6版本引入了Dircet模式。



Driect模式就是将kafka看成存数据的一方，这种模式没有采用Receiver接收器模式，而是采用直连的方式，不是被动接收数据，而是主动去取数据，当任务失败后代码中如果设置了checkpoint目录，那么最近消费Kafka批次信息也会保存在checkpoint中。当SparkStreaming停止后，我们可以使用val ssc = StreamFactory.getOrCreate(checkpointDir,Fun)来恢复停止之前SparkStreaming处理数据的进度，当然，这种方式存在重复消费数据和逻辑改变之后不可执行的问题。

Direct模式底层读取Kafka数据实现是Simple Consumer api实现，这种api提供了从每批次数据中获取offset的接口，所以对于精准消费数据的场景，可以使用Direct 模式手动维护offset方式来实现数据精准消费。

此外，Direct模式的并行度与当前读取的topic的partition个数一致，所以Direct模式并行度由读取的kafka中topic的partition数决定的。

***Direct模式**如何保证消费Kafka数据offset精准性？*

1. **checkpoint管理**

如果设置了checkpoint ,那么最近消费批次数据会存储在checkpoint中。这种有缺点: 第一，当从checkpoint中恢复数据时，有可能造成重复的消费。第二，当代码逻辑改变时，无法从checkpoint中来恢复offset.

2. **依赖Kafka存储**

依靠kafka 来存储消费者offset,kafka 中有一个特殊的topic 来存储消费者offset。新的消费者api中，会定期自动提交offset。这种情况有可能也不是我们想要的，因为有可能消费者自动提交了offset,但是后期SparkStreaming 没有将接收来的数据及时处理保存。这里也就是为什么会在配置中将enable.auto.commit 设置成false的原因。这种消费模式也称最多消费一次（at-most-once），默认sparkStreaming 拉取到数据之后就可以更新offset,无论是否消费成功，自动提交offset的频率由参数auto.commit.interval.ms 决定，默认5s。

如果我们能保证完全处理完业务之后，可以后期异步的手动提交消费者offset。但是这种将offset存储在kafka中由参数offsets.retention.minutes=1440控制是否过期删除，默认是保存一天，如果停机没有消费达到时长，存储在kafka中的消费者组会被清空，offset也就被清除了。

3. **手动维护**

自己存储offset,这样在处理逻辑时，保证数据处理的事务，如果处理数据失败，就不保存offset，处理数据成功则保存offset.这样可以做到精准的处理一次处理数据。

1.6 Flink读取Kafka中数据如何保证消费一致性？

1.6.1 Flink消费Kafka 数据offset维护方式

Flink提供了消费kafka数据的offset如何提交给Kafka或者zookeeper(kafka0.8之前)的配置。注意，Flink并不依赖提交给Kafka或者zookeeper中的offset来保证容错。提交的offset只是为了外部来查询监视kafka数据消费的情况。

配置offset的提交方式取决于是否为job设置开启checkpoint。可以使用env.enableCheckpointing(5000)来设置开启checkpoint。

- 关闭checkpoint：

如果禁用了checkpoint，那么offset位置的提交取决于Flink读取kafka客户端的配置，enable.auto.commit（auto.commit.enable【Kafka 0.8】)配置是否开启自动提交offset, auto.commit.interval.ms决定自动提交offset的周期。

- 开启checkpoint：

如果开启了checkpoint，那么当checkpoint保存状态完成后，将checkpoint中保存的offset位置提交到kafka。这样保证了Kafka中保存的offset和checkpoint中保存的offset一致，可以通过配置setCommitOffsetsOnCheckpoints(boolean)来配置是否将checkpoint中的offset提交到kafka中（默认是true）。如果使用这种方式，那么properties中配置的kafka offset自动提交参数enable.auto.commit和周期提交参数auto.commit.interval.ms参数将被忽略。

1.6.2 使用checkpoint+两阶提交保证仅一次消费Kafka中数据

当谈及“exactly-once semantics”仅一次处理数据时，指的是每条数据只会影响最终结果一次。Flink可以保证当机器出现故障或者程序出现错误时，也没有重复的数据或者未被处理的数据出现，实现仅一次处理的语义。Flink开发出了checkpointing机制，这种机制是在Flink应用内部实现仅一次处理数据的基础。

checkpoint中包含：

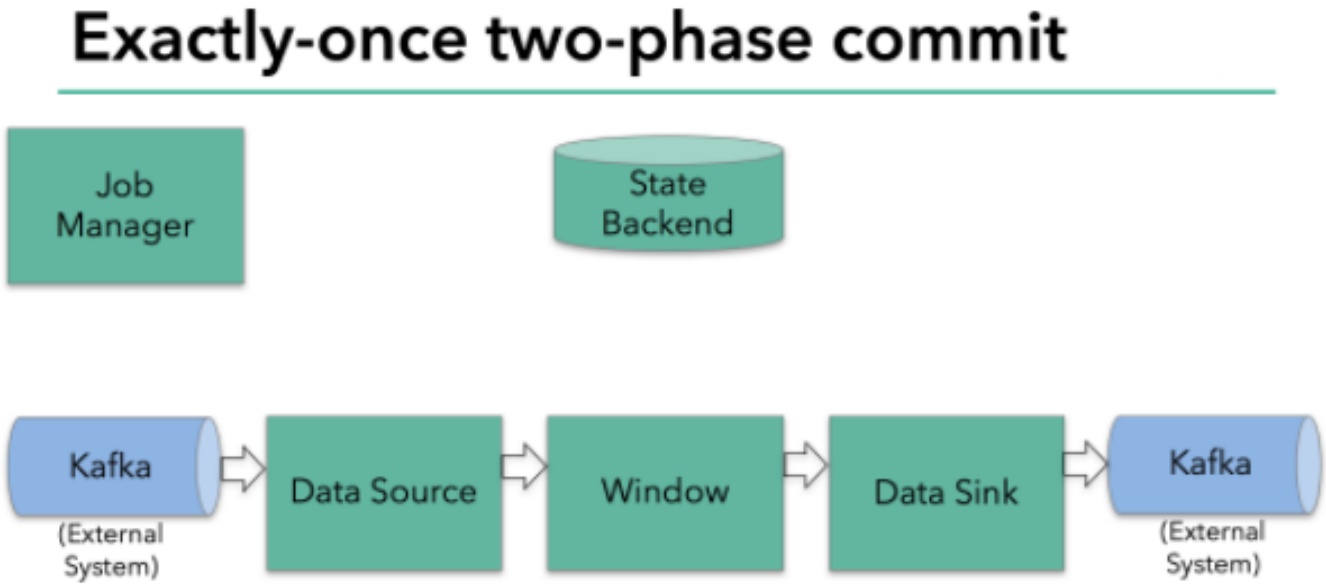
1. 当前应用的状态
2. 当前消费流数据的位置

在Flink1.4版本之前，Flink仅一次处理数据只限于Flink应用内部（可以使用checkpoint机制实现仅一次数据数据语义），当Flink处理完的数据需要写入外部系统时，不保证仅一次处理数据。为了提供端到端的仅一次处理数据，在将数据写入外部系统时也要保证仅一次处理数据，这些外部系统必须提供一种手段来允许程序提交或者回滚写入操作，同时还要保证与Flink的checkpoint机制协调使用。

在分布式系统中协调提交和回滚的常见方法就是两阶段提交协议。下面给出一个实例了解Flink如何使用两阶段提交协议来实现数据仅一次处理语义。

该实例是从kafka中读取数据，经过处理数据之后将结果再写回kafka。kafka0.11版本之后支持事务，这也是Flink与kafka交互时仅一次处理的必要条件。【注意：当Flink处理完的数据写入kafka时，即当sink为kafka时，自动封装了两阶段提交协议】。Flink支持仅一次处理数据不仅仅限于和Kafka的结合，只要sink提供了必要的两阶段协调实现，可以对任何sink都能实现仅一次处理数据语义。

其原理如下：

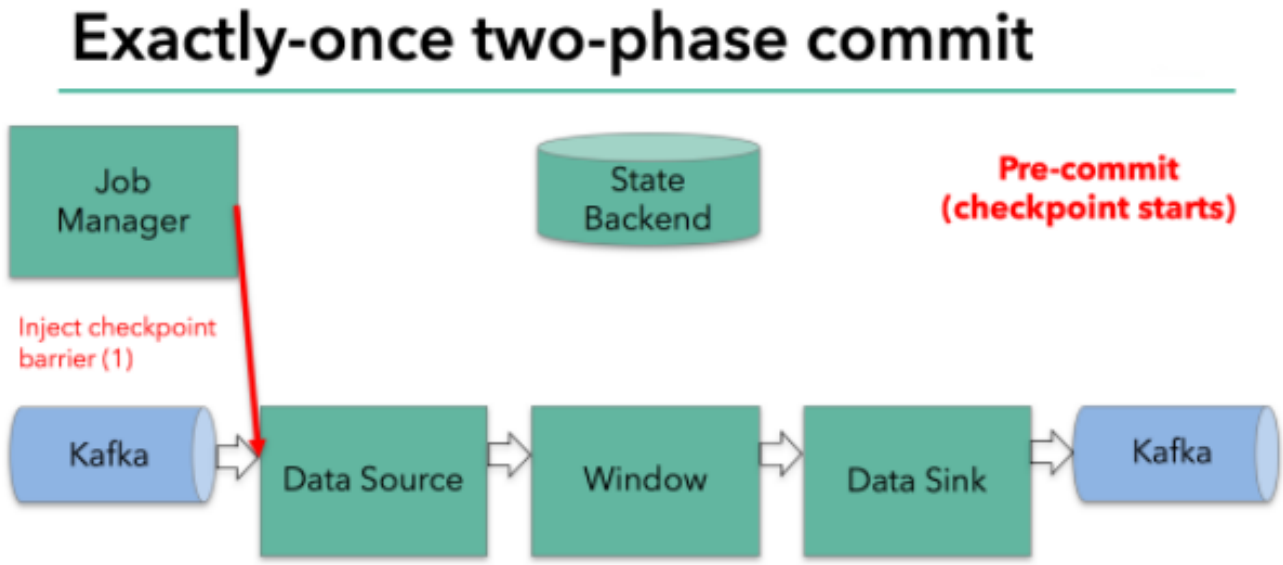


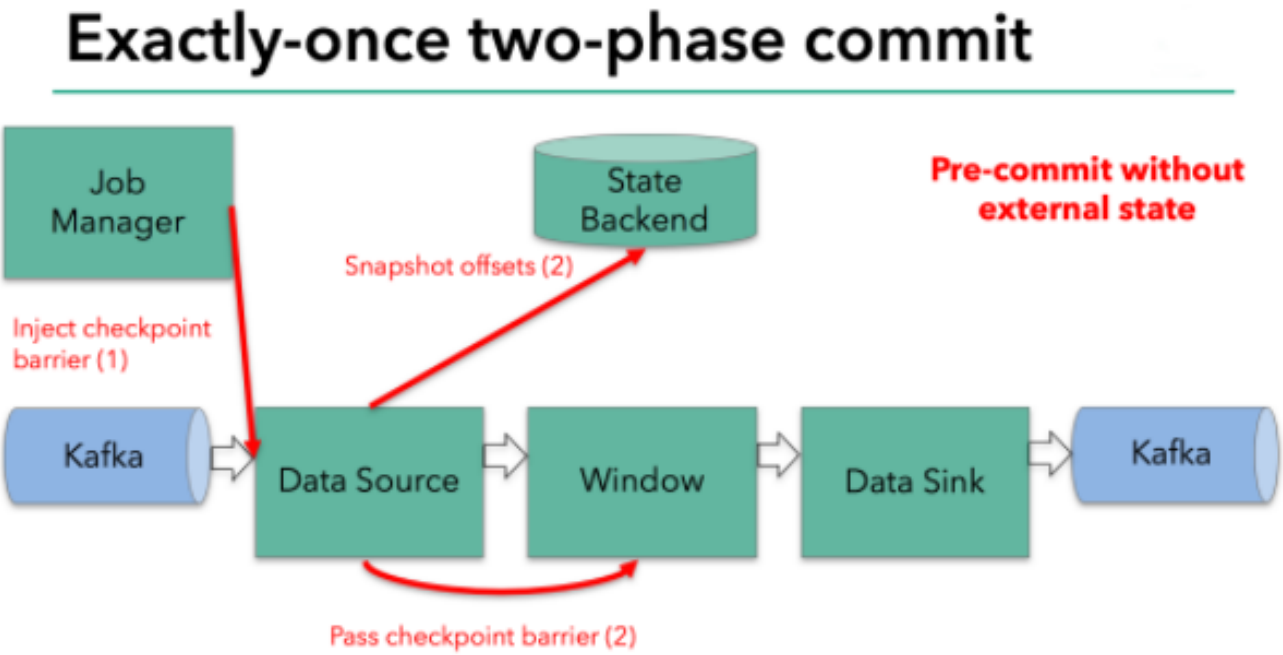
上图Flink程序包含以下组件：

1. 一个从kafka中读取数据的source
2. 一个窗口聚合操作
3. 一个将结果写往kafka的sink。

要使sink支持仅一次处理数据语义，必须以事务的方式将数据写往kafka,将两次checkpoint之间的操作当做一个事务提交，确保出现故障时操作能够被回滚。假设出现故障，在分布式多并发执行sink的应用程序中，仅仅执行单次提交或回滚事务是不够的，因为分布式中的各个sink程序都必须对这些提交或者回滚达成共识，这样才能保证两次checkpoint之间的数据得到一个一致性的结果。Flink使用两阶段提交协议(pre-commit+commit)来实现这个问题。

Filnk checkpointing开始时就进入到pre-commit阶段， 具体来说，一旦checkpoint开始，Flink的JobManager向输入流中写入一个checkpoint barrier将流中所有消息分隔成属于本次checkpoint的消息以及属于下次checkpoint的消息，barrier也会在操作算子间流转，对于每个operator来说，该barrier会触发operator的State Backend来为当前的operator来打快照。如下图示：

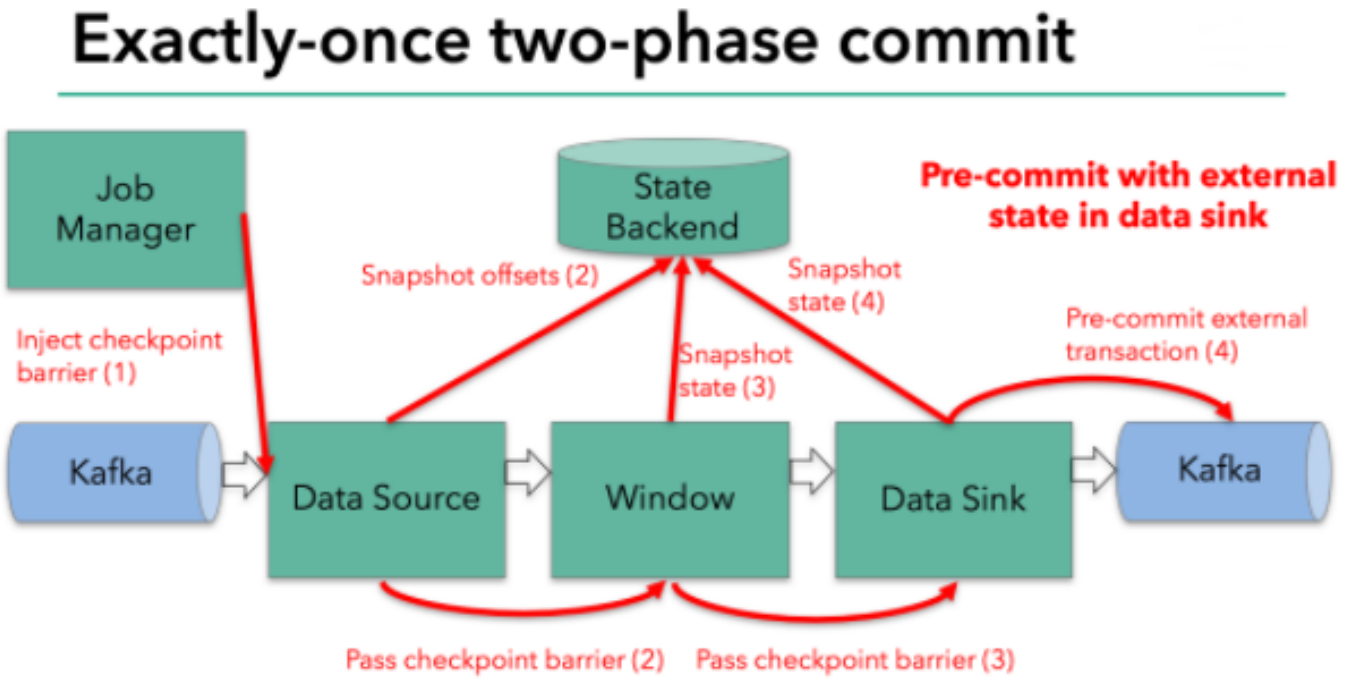




Flink DataSource中存储着Kafka消费的offset，当完成快照保存后，将chechkpoint barrier传递给下一个operator。这种方式只有在Flink内部状态的场景是可行的，内部状态指的是由Flink的State Backend管理状态，例如上面的window的状态就是内部状态管理。只有当内部状态时，pre-commit阶段无需执行额外的操作，仅仅是写入一些定义好的状态变量即可，checkpoint成功时Flink负责提交这些状态写入，否则就不写入当前状态。

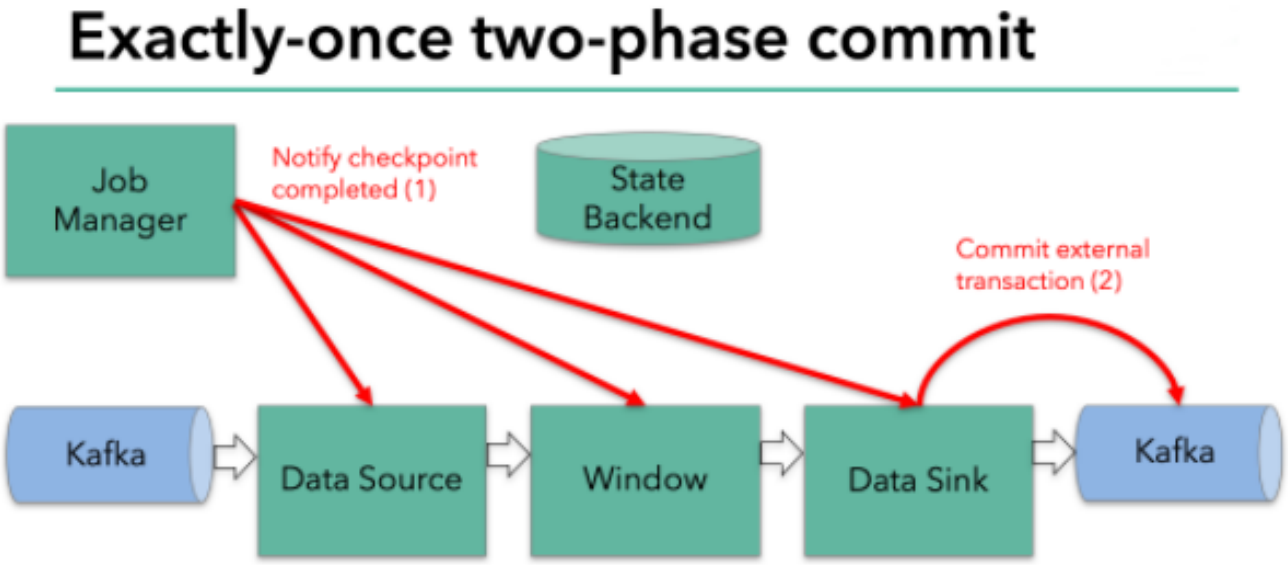
但是，一旦operator操作包含外部状态，事情就不一样了。我们不能像处理内部状态一样处理外部状态，因为外部状态涉及到与外部系统的交互。这种情况下，外部系统必须要支持可以与两阶段提交协议绑定的事务才能保证仅一次处理数据。

本例中的data sink是将数据写往kafka，因为写往kafka是有外部状态的，这种情况下，pre-commit阶段下data sink 在保存状态到State Backend的同时，还必须pre-commit外部的事务。如下图：



当checkpoint barrier在所有的operator都传递一遍切对应的快照都成功完成之后，pre-commit阶段才算完成。这个过程中所有创建的快照都被视为checkpoint的一部分，checkpoint中保存着整个应用的全局状态，当然也包含pre-commit阶段提交的外部状态。当程序出现崩溃时，我们可以回滚状态到最新已经完成快照的时间点。

下一步就是通知所有的operator，告诉它们checkpoint已经完成，这便是两阶段提交的第二个阶段：commit阶段。这个阶段中JobManager会为应用中的每个operator发起checkpoint已经完成的回调逻辑。本例中，DataSource和Winow操作都没有外部状态，因此在该阶段，这两个operator无需执行任何逻辑，但是Data Sink是有外部状态的，因此此时我们需要提交外部事务。如下图示：



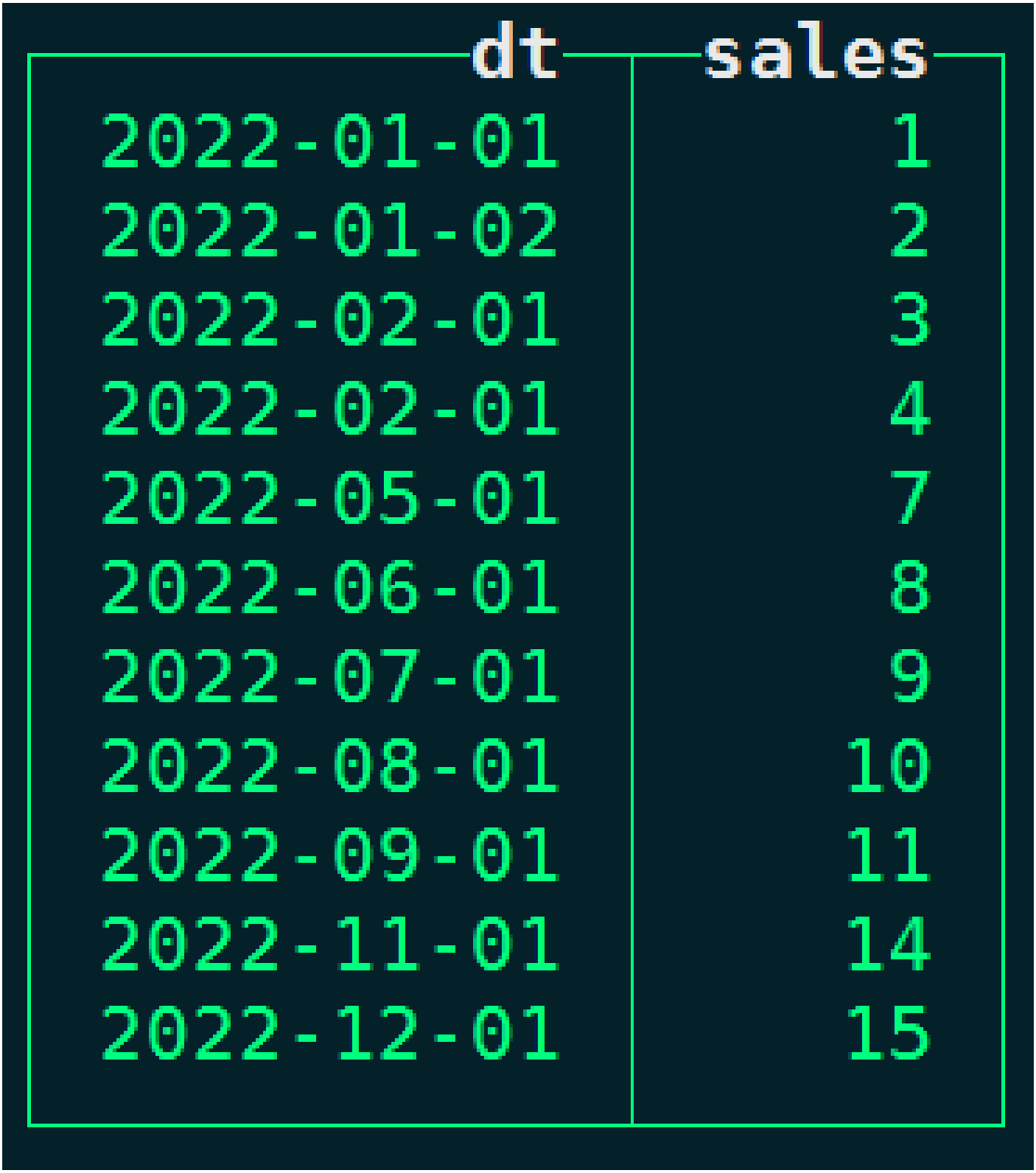
汇总以上信息，总结得出：

- 一旦所有的operator完成各自的pre-commit,他们会发起一个commit操作。
- 如果一个operator的pre-commit失败，所有其他的operator 的pre-commit必须被终止，并且Flink会回滚到最近成功完成的checkpoint位置。
- 一旦pre-commit完成，必须要确保commit也要成功，内部的operator和外部的系统都要对此进行保证。假设commit失败【网络故障原因】，Flink程序就会崩溃，然后根据用户重启策略执行重启逻辑，重启之后会再次commit。

因此，所有的operator必须对checkpoint最终结果达成共识，即所有的operator都必须认定数据提交要么成功执行，要么被终止然后回滚。

1.7 SQL-在Clickhouse中实现如下业务

Clickhouse中有如下销售额表sale_info，对应的列为dt-日期，sales-销售额:



dt	sales
2022-01-01	1
2022-01-02	2
2022-02-01	3
2022-02-01	4
2022-05-01	7
2022-06-01	8
2022-07-01	9
2022-08-01	10
2022-09-01	11
2022-11-01	14
2022-12-01	15

根据以上表统计全年各个月的累计销售额，没有的月份补齐月份，结果如下：

dt	total_sales
202201	3
202202	10
202203	10
202204	10
202205	17
202206	25
202207	34
202208	44
202209	55
202210	55
202211	69
202212	84

准备sale_info和数据，sale_info建表及插入数据如下：

```
CREATE TABLE sale_info
(
    `dt` date,
    `sales` Int32
)
ENGINE = MergeTree()
```

```
ORDER BY dt;

insert into sale_info values
('2022-01-01',1),
('2022-01-02',2),
('2022-02-01',3),
('2022-02-01',4),
('2022-05-01',7),
('2022-06-01',8),
('2022-07-01',9),
('2022-08-01',10),
('2022-09-01',11),
('2022-11-01',14),
('2022-12-01',15);
```

为了补齐各个月份，这里需要创建一个月份基础表month_info，方便进行关联。month_info表及数据如下：

```
CREATE TABLE month_info
(
    `dt` date
)
ENGINE = MergeTree()
ORDER BY dt;

insert into month_info values
('2022-01-31'),
('2022-02-28'),
('2022-03-31'),
('2022-04-30'),
('2022-05-31'),
('2022-06-30'),
('2022-07-31'),
('2022-08-31'),
('2022-09-30'),
('2022-10-31'),
('2022-11-30'),
('2022-12-31');
```

实现以上需求步骤如下:

1. 将sale_info 与 基础表month_info 进行关联，补全月份信息
2. 使用sum() over() 窗口函数进行统计累加

最终SQL及结果如下：

```
SELECT DISTINCT
    toYYYYMM(dt) AS dt,
    sum(sales) OVER (PARTITION BY 1 ORDER BY dt ASC) AS total_sales
FROM
(
    SELECT
        b.dt AS dt,
        a.sales AS sales
    FROM sale_info AS a
    RIGHT JOIN month_info AS b ON toYYYYMM(a.dt) = toYYYYMM(b.dt)
)
```

1.8 SQL-解析嵌套的数组类型

需求：Hive中有一张表t1,数据如下:

```
1       zs1     [1,2,[3,4,5],[6,7,8],9]
2       zs2     [10,20,[30,40,50],[60,70,80],90]
3       zs3     [100,200,[300,400,500],[600,700,800],900]
4       zs4     [1000,2000,[3000,4000,5000],[6000,7000,8000],9000]
```

要求将infos列中的数据，按照逗号划分得到如下数据结果：

```
1       zs1     1
1       zs1     2
1       zs1     [3,4,5]
1       zs1     [6,7,8]
1       zs1     9
2       zs2     10
2       zs2     20
2       zs2     [30,40,50]
2       zs2     [60,70,80]
2       zs2     90
3       zs3     100
3       zs3     200
3       zs3     [300,400,500]
3       zs3     [600,700,800]
3       zs3     900
4       zs4     1000
4       zs4     2000
4       zs4     [3000,4000,5000]
4       zs4     [6000,7000,8000]
4       zs4     9000
```

数据data.txt(数据按照'\t'分开):

```
1	zs1	[1,2,[3,4,5],[6,7,8],9]
2	zs2	[10,20,[30,40,50],[60,70,80],90]
```

```
3	zs3	[100,200,[300,400,500],[600,700,800],900]
4	zs4	[1000,2000,[3000,4000,5000],[6000,7000,8000],9000]
```

创建Hive表并加载数据：

```
create table t1 (id int ,name string,infos String) row format delimited fields terminated by '\t';
load data local inpath '/root/data.txt' into table t1;
```

分析：以上数据在Hive中不大可能使用一条HQL能将结果处理成最终结果，这里需要使用自定义函数进行解析处理，思路如下：

1. 利用正则将字符串中“[数字,数字,数字]” 字符串匹配出来
2. 匹配后，将“[数字,数字,数字]” 改成 “[数字-数字-数字]”,再更新到对应的字符串
3. 针对修改后的字符串，按照 “，” 进行分割
4. 将分割后的字符串使用explode 函数，一变多处理
5. 对新列中“[数字-数字-数字]”数据使用 regexp_replace 函数处理成原来 “[数字,数字,数字]”

SparkSQL代码如下：

```
object ReadHiveData {
  def main(args: Array[String]): Unit = {
    val spark: SparkSession = SparkSession.builder().master("local")
      .appName("test")
      .config("hive.metastore.uris", "thrift://node1:9083")
      .enableHiveSupport()
      .getOrCreate()

    spark.sparkContext.setLogLevel("Error")

    spark.table("t1").createTempView("temp")
    spark.sql("select * from temp " ).show(false)

    spark.udf.register("myudf",(str:String)=>{
      //1.准备正则表达式，匹配 “[数字,数字,数字]”
      //[0-9] :匹配所有的数字；[0-9,] 匹配所有数字和逗号 ，[0-9,]+匹配多次
      val pattern: Pattern = Pattern.compile("\\[[0-9,]+\\]")
      //2.匹配正则表达式
      val matcher: Matcher = pattern.matcher(str)

      //3.去掉字符串的开头“[” 和结尾 “]”
      var returnString = str.stripPrefix("[").stripSuffix("]")

      //4.循环多次找到匹配结果
      while(matcher.find()){
```

```
        //5.匹配到的符合正则表达式的结果
        val matchStr: String = matcher.group
        //6.将原有字符串中匹配到的 “[数字,数字,数字]” 字符串中的“,”替换成“-”
        returnString = returnString.replace(matchStr,matchStr.replace(",","-"))
      }
      returnString
    })

    val result:DataFrame = spark.sql(
      """
        | select
        |   id,name,regexp_replace(str,"-",",") as new_str
        | from
        |  (select id,name,explode(split(myudf(infos),",")) as str from temp ) as trans_tbl
      """.stripMargin)
    //打印结果
    result.show()
    //结果写出到Hive表中
    result.write.mode(SaveMode.Overwrite).saveAsTable("result")
  }
}
```