# 数仓案例

## 目标

在学习数据仓库的基本理论知识和所需的工具后，通过案例将整个数据仓库的开发流程串联起来，掌握数据仓库从模型设计、物理实现、加载数据、自动同步以及分析指标和的过程，了解报表展示。

## 业务系统

### 业务场景

业务场景我们使用数仓的经典的销售订单源系统,业务逻辑很简单，有两个基本信息表产品表和客户表，产品表记录产品名称、编号和分类,客户表记录客户编号,客户名称以及其他基本信息，一个业务表订单表，记录哪个客户购买了什么产品，以及花费多少。

### 数据库设计

### 关系数据模型

1970年美国IBM公司SanJose研究室的研究员E.F.Codd首次提出了数据库系统的关系模型，开创了数据库的关系方法和关系数据理论的研究，为数据库技术奠定了理论基础。[关系数据库系统](https://baike.baidu.com/item/%E5%85%B3%E7%B3%BB%E6%95%B0%E6%8D%AE%E5%BA%93%E7%B3%BB%E7%BB%9F/9797711" \t "_blank)是支持[关系模型](https://baike.baidu.com/item/%E5%85%B3%E7%B3%BB%E6%A8%A1%E5%9E%8B/3189329" \t "_blank)的[数据库系统](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E5%BA%93%E7%B3%BB%E7%BB%9F/215176" \t "_blank)。常见的数据库MySQL，Oracle、DB2，SQLServer都是关系型数据库。

#### 关系：

由行和列构成的二维结构，对应关系数据中表，也就是数据的存储结构

#### 属性及属性域

属性由属性名和类型构成的顺序对，对应关系数据库中表的列，在关系型数据模型中，关系对应着表，表中的行对应不同的记录，表中的列对应不同的属性，属性是顺序无关的，以任何顺序出现，关系都保持不变，也就是说表中的列是没有顺序的。

每一个属性都有一个预定义的范围，也就是属性的取值范围，它允许我们定义属性可以具有的值的意义，并且可以拒绝不合理的操作，比如月份属性是1-12之间的整形数值。

#### 元组

关系数据库中的一个表的行，也就是一条记录。

#### 关系数据库

关系数据库就是一系列规范化的表的集合，这里的规范化可以理解为表结构的正确性。也就是我们通常所说的范式。

关系数据模型中术语是“关系、属性、元组”跟“表、列、行”，在关系数据库中一般称作“表、列、行”或者“实体、字段、记录”等。

关系表具有一些特性

* 在一个数据库中，表名称唯一
* 表中的每个列名称不同，不同的表列名称可以相同
* 列是无序的
* 行是无序的

#### 关系数据模型中的键

关系模型中具有一些键：超键、候选键、主键、外键，这些键用来约束关系完整性。

超键：super key 在关系中能唯一标识元组的属性集合，可能含有唯一标识记录的不必要的额外列。通常我们只对唯一标识记录的最少数量的列感兴趣，也就是主键。

候选键：candidate key 不含多余属性的超键，去掉一个属性就不是超键，就是候选键候选键能唯一标识该记录，并且是超键的最小子集，候选键的值不允许为空。一个表可以有多个候选键。

主键：从候选键中确定一个唯一标识表中记录的键作为主键，其他的没有被选做主键的候选键称为备用键。主键的选择不当会引起性能问题，主键的选择遵循以下原则：

* 主键要尽可能的小
* 主键值不应该被改变，主键通常会被其他表所引用，如果更改后，引用的键值也需要做相应的更改，否则引用就无效了
* 主键通常使用数值类型
* 最好是单列

一般在表的设计中，有能唯一标识记录的字段，但是跟业务有关，我们一般不用作主键，额外增加一个自增的列做为主键是一个不错的选择。我们在设计表的时候必须给表一个主键，如果没有主键表中的数据更新和删除就会很困难。

聚集索引和主键的区别和联系：聚集索引时表中的数据的物理存储顺序，每个表只能有一个聚集索引，聚集索引可以不要求具有唯一性。有的数据库中如果表没有聚集索引，在创建主键的时候可以设定把主键作为聚集索引。

#### 关系完整性

上面我们讲了关系模型中的键，这些键约束了完整性的规则，关系数据模型有两个重要的完整性规则：实体完整性和参照完整性。

实体完整性即表中的主键列不能为空。

NULL：空值，表示目前还不知道或者不可用的值，

空值参与算术表达式结果为NULL

true and null=null

fasle and null = false

null and null =null

true or null = true

false or null = null

null or null =null

not null = null

null=null 会返回null 而不是ture，如果要判断是否为不为空，要使用 is null 或者is not null

参照完整性：就是表中存在外键，外键必须与主表的某些记录的候选键相同，或者外键的值必须全部为空。

除上述的完整性规则，通常实际的业务场景有业务规则的约束，约束特定的列能够接受的值，一般的数据库都有，即check约束。比如性别取值：男和女等,这些约束要根据业务来制定。

#### 规范化，范式

规范化是组织数据的一种技术，规范化方法对表进行分解，以消除数据冗余，避免异常更新，提高数据完整性。规范化是通过范式规则实现的，最常用的范式有：第一范式（1NF）、第二范式（2NF）、第三范式（3NF）。

* 第一范式1NF，表中的列只能含有原子性的值，也就是每个列的值都不能再分，假如有一个人有两个手机号，不能存到一个列中，手机号需要单独存一个表中。
* 第二范式2NF，2NF要满足两个条件：

1. 满足第一范式
2. 没有部分依赖，即每个非主属性完全函数依赖于候选键。

也就是表要有一个主键，其他字段都依赖主键，每个非主属性是由整个主键函数决定的，而不能由主键的一部分来决定，如果有属性跟主键不相关则叫做部分依赖，就不满足第二范式。

例如(学生选课表):

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 学生 | 课程 | | 教师 | 教材 | 教室 | 上课时间 | 职称 |
| 李四 | Spring | | 张老师 | 《Spring深入浅出》 | ３０１ | 08:00 | 副教授 |
| 张三 | Struts | 杨老师 | | 《Struts in Action》 | ３０２ | 13:30 | 教授 |

这里通过（学生，课程）可以确定教师、教师职称，教材，教室和上课时间，所以可以把（学生，课程）作为主键。但是，教材并不完全依赖于（学生，课程），只拿出课程就可以确定教材，因为一个课程，一定指定了某个教材。这就叫不完全依赖，或者部分依赖。出现这种情况，就不满足第二范式。

修改后如下：

选课表：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 学生 | 课程 | | 教师 | 教室 | 职称 | 上课时间 |
| 李四 | Spring | | 张老师 | ３０１ | 副教授 | 08:00 |
| 张三 | Struts | 杨老师 | | ３０２ | 教授 | 13:30 |

课程表：

|  |  |
| --- | --- |
| 课程 | 教材 |
| Spring | 《Spring深入浅出》 |
| Struts | 《Struts in Action》 |

* 第三范式

第三范式也要满足两个条件：

1. 满足第二范式
2. 没有传递依赖，所有非主属性对任何候选关键字都不存在传递依赖。

简单的说，第三范式首先要满足第二范式，其次非主属性之间不存在函数依赖。由于满足了第二范式，表示每个非主属性都函数依赖于主键。如果非主属性之间存在了函数依赖，就会存在传递依赖，这样就不满足第三范式。

还是上面选课表例中修改后的选课表中，一个教师能确定一个教师职称。这样，教师依赖于（学生，课程），而教师职称又依赖于教师，这叫传递依赖。第三范式就是要消除传递依赖。

满足第三范式的选课表如下：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 学生 | 课程 | | 教师 | 教室 | 上课时间 |
| 李四 | Spring | | 张老师 | ３０１ | 08:00 |
| 张三 | Struts | 杨老师 | | ３０２ | 13:30 |

课程表：

|  |  |
| --- | --- |
| 课程 | 教材 |
| Spring | 《Spring深入浅出》 |
| Struts | 《Struts in Action》 |

教师表

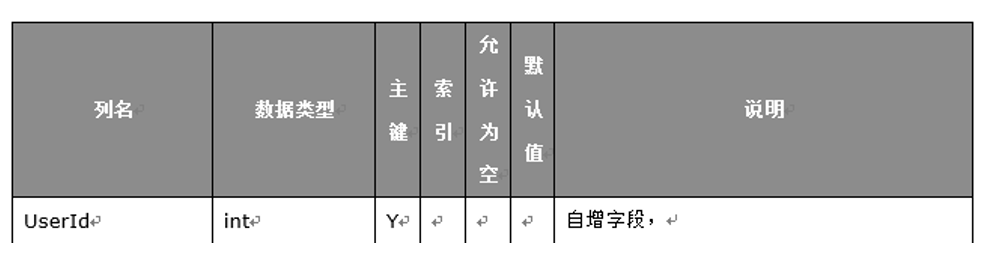
|  |  |
| --- | --- |
| 教师 | 教师职称 |
| 张老师 | 副教授 |
| 杨老师 | 教授 |

范式主要有六种：第一范式、第二范式、第三范式、BC范式、第四范式和第五范式，一般设计数据库的时候满足三范式就可以了，规范化带来的好处是通过减少数据冗余提高更新数据的效率，同时保证数据的完整性。然而，我们在实际的应用中也要防止过度规范化，规范化的程度越高，划分的表就越多，在查询数据时要关联的表越多，就会影响查询的效率。关键的问题是要依据业务需求，仔细权衡数据查询和数据更新的关系，制定最合适的规范化程度，还有一点需要注意的是，不要为了遵循严格的规范化规则而修改业务需求。

#### mysql数据库设计规范

##### 命名规范

1. 数据库设计，需要先设计逻辑模型，再设计物理模型，最后形成各模型的模型文件和数据库设计文档，文档包括：表，字段，字段类型主键、默认值和说明



1. 所有的数据库对象名称必须使用小写字母并用下划线表示。
2. 所有数据库对象名称禁止使用mysql保留关键字
3. 数据库对象的命名要能做到见名知意，并且最好不要超过32个字符。太长不方便使用，并且会在传输时增加网络开销
4. 临时表必须以tmp\_为前缀并以日期为后缀
5. 备份表必须以bak\_为前缀并以日期为后缀
6. 所有存储相同数据的列名和列类型必须一致，比如user表中的id和order表中的user\_id

##### 数据库设计规范

1. 数据库读写分开，做主备，主库主要进行写操作，备库主要进行查询操作。主库采用Innodb，备库采用myisam库。
2. 数据库和表的字符集统一使用UTF-8，如果要存储一些如表情符号的，还需使用UTF-8的拓展字符集, 数据库，表，字段字符集一定要统一，统一字符集可以避免由于字符集转换产生的乱码, 在mysql中UTF-8字符集，汉字占3字节，ASCII码占1字节.
3. 所有表和字段都需要添加注释
4. 数据库设计文档必须要有
5. 尽量控制单表数据量大小，
   1. 建议控制在500万以内，虽然500万并不是mysql的数据库限制，但是会给修改表结构，备份，恢复带来很大困难。
   2. 单表可存储数据量大小取决于存储设置和文件系统
   3. 想减少单表数据量：历史数据归档(常见于日志表)，分库分表(常见于业务表)，分区表
   4. 建议不要使用mysql分区表，因为分区表在物理上表现为多个文件，在逻辑上表现为一个表。如果一定要分区，请谨慎选择分区键，跨分区查询效率比查询大数据量的单表查询效率更低
   5. 建议采物理分表的方式管理大数据，但是对应用程序的开发要求和复杂度更高
6. 尽量做到冷热数据分离，减少表的宽度(字段数)

减少磁盘IO，保证热数据的内存缓存命中率，更有效的利用缓存，避免读入无用的冷数据，对表的列进行拆分，将经常使用的列放到一个表中，可以避免过多的关联操作，也可以提高查询性能。

1. 禁止在表中建立预留字段 ，

a) 预留字段很难做到见名知义，预留字段无法确定存储的数据类型，后期如果修改字段类型，会对全表锁定，严重影响数据库的并发性

b) 对目前mysql来说，修改一个字段的成本要远远大于增加一个字段的成本

1. 禁止在数据库中存储图片，文件等二级制数据

a) 这类数据如果要存，就得使用blob或者text这样的大字段加以存储，会影响数据库的性能

b)文件这种通常所占数据容量很大，会在短时间内造成数据库文件的快速增长，而数据库在读取数据时，会进行大量的随机IO操作，如果数据文件过大，IO操作会非常耗时，从而影响数据库性能

c) 正确做法是将这类数据存储在文件服务器中，而数据库只存储地址信息

1. 禁止在线上做数据库压力测试

a) 会对正常业务造成影响，也会产生很多垃圾数据

b) 建议建立专门的压力测试数据库，进行测试，然后对比测试服务器和线上服务器的硬件环境，评估线上数据库的性能

1. 禁止从开发环境，测试环境直连生产环境数据库

##### 索引设计规范(Innodb中主键实质上是一个索引)

1. 限制每张表上索引数量，建议单表不超过5个索引。索引并不是越多越好，可以提高查询效率，但是会降低插入和更新的效率。甚至在一些情况下，还会降低查询效率，因为mysql优化器在选择如何优化查询时，会根据统计信息，对每一个可用索引来进行评估，以生成一个最好的执行计划，如果同时有很多索引都可以用于查询，就会增加mysql查询优化器生成查询计划的时间。
2. 每个Innodb表都必须有一个主键。Innodb是一种索引组织表，是指数据存储的逻辑顺序和索引的顺序是相同，Innodb是按照主键索引的顺序来组织表的，因此，每个Innodb表都必须要有一个主键，如果我们没有指定主键，那么Innodb会优先选择表中第一个非空唯一索引来作为主键，如果没有这个索引，那么Innodb会自动生成一个占6字节的主键，而这个主键的性能并不是最好。
3. 不使用更新频繁的列作为主键，不使用多列联合主键。因为Innodb是一种索引索引组织表，如果主键上的值频繁更新，就意味着数据存储的逻辑顺序频繁变动，必然会带来大量的IO操作，降低数据库性能
4. 不要使用uuid，md5，hash，字符串列作为主键。因为这种主键不能保证主键的值是顺序增长的，如果后来的主键值在已有主键值的中间段，那么这个主键插入的时候，会将所有主键值大于它的列都向后移。
5. 最好选择能保证值的顺序为顺序增长的列为主键。并且数据不能重复，每个表给设置一个自增的id做主键

6. 哪些列上建立索引?

a） 在select，delete，update的where从句中的列

b) 包含在order by，group by，distinct字段中的列

c) 多表join的关联列：mysql对关联操作的处理方式只有一种，那就是嵌套循环的关联方式，所以这种操作的性能对关联列上的索引的依赖性很大

7. 复合索引，如何选择索引列的顺序?

a) 从左到右的顺序来使用的

1. 区分度(列中group by的数目和此列总行数的比值趋近于1)最高的列放在联合索引的最左侧
2. 在区分度差不多的情况下，尽量把字段长度小的放在联合索引的最左侧，因为同样的行数，字段小的文件也小，读取时IO性能更优
3. 使用最频繁的列放在联合索引的左侧，这样的话，可以较少地建立索引就能满足需求
4. 避免建立冗余索引和重复索引
5. 对于频繁的查询优先使用覆盖索引 ,就是包含了所有查询字段的索引，这样可以避免Innodb表进行索引的二次查找，并可以把随机IO变为顺序IO提高查询效率
6. 尽量避免使用外键,模型设计的时候要有外键关系，建表的是要去掉外键
   1. mysql和别的数据库不同，会自动在外键上建立索引，会降低数据库的写性能
   2. 建议不使用外键约束，但是一定要在表与表之间的关联键上建立索引，虽然外键是为了保证数据的完整性，但是最好在代码中去保证。

##### 字段设计规范

1. 优先选择符合存储需要的最小的数据类型

a) 尽量将字符串转化为数字类型存储：如将ip存储为数字：inet\_aton(‘255.255.255.255’) = 4294967295 ,反之， inet\_ntoa(4294967295) = ‘255.255.255.255’

b) 对于非负整型数据，优先使用无符号整型来存储，如：id,age,无符号相对于有符号，可以多出一倍的存储空间

c) mysql中，varchar(n)中n表示字符数而不是字节数

d) 避免使用text，blob来存储字段，这种类型只能使用前缀索引，如果非要使用，建议将这种数据分离到单独的拓展表中

e) 能够确定范围的值能用tinyint的就不要用int

f) 自增的主键字段使用bigint

3.避免使用enum类型。枚举本身是一个字符串类型，但是内部确是用正数类型来存储的，所以最多可存储65535种不同的值，修改的话必须使用alter语句，直接修改元数据，有操作风险；order by效率低，必须转换并无法使用索引，禁止使用数值作为enum值，因为enum本身是索引顺序存储的，会造成逻辑混淆

4. 尽可能把所有列定义为not null。

a) 索引null列需要额外的空间来保存，占更多空间

b) 进行比较和计算时，对null值作特别的处理，可能造成索引失效

5. 禁止使用字符串来存储日期型数据。

a) 无法使用日期函数计算比较

b)字符串存储要占更多的内存空间，datetime(8字节)和timestamp(本身是以int存储，占4字节,范围:1970-01-01 00:00:01到2038-01-19 03:14:07)

d)可以使用数值存储日期

6. 财务相关数据，使用decimal类型 (精准浮点类型，在计算时不丢失精度)。

##### SQL开发规范

1. 建议使用预编译语句(prepareStatment)进行数据库操作

a) 可以同步执行预编译计划，减少预编译时间

b) 可以有效避免动态sql带来的SQL注入的问题

c) 只传参数，一次解析，多次使用，比传递sql语句更高效

2. 避免数据类型的隐式转换

a) 一般出现在where从句中，会导致索引失效，如：select id,name from user where id = ‘12’;

3. 充分利用已存在的索引

a) 避免使用双%的查询条件，不走索引

b) 一个SQL只能利用到复合索引中的一列进行范围查询

c) 使用left join或not exists来优化not in操作

4. 程序连接不同的数据库使用不同的账号，禁止跨库查询

a) 为数据库迁移和分库分表留出余地

b) 降低业务耦合度

c) 避免权限过大而产生的安全风险

5. 禁止使用select \* 来查询，必须用字段名

a) 可能会消耗更多的cpu和IO以及网络资源

b) 无法使用覆盖索引

c) 可以减少表结构变更对已有程序的影响

6. 禁止使用不含字段列表的insert语句。 可以减少表结构变更对已有程序的影响

7. 禁止使用子查询

a) 虽然可使sql可读性好，但是缺点远远大于优点

b) 子查询返回的结果集无法使用索引，结果集会被存储到一个临时表中，结果集越大性能越低 c)把子查询优化为join操作，但是并不是所有的都可以优化为join，一般情况下，只有当子查询是在in字句中，并且子查询是一个简单的sql(不包含union，group by，order by，limit)才能转换为关联查询

8. 避免join过多的表

a) 每join一个表会占一部分内存(join\_buffer\_size)

b) 会产生临时表操作，影响查询效率

c) mysql最多允许关联61个表，建议不超过5个

9. 减少同数据库的交互次数

a)数据库更适合处理批量操作

b)合并多个相同的操作到一起，提高处理效率

10. 使用in代替or

a) in的值不要超过500个

b) in 操作可以有效利用索引

11. 禁止使用order by rand()进行随机排序

a) 会把表中所有符合条件的数据装载到内存中进行排序

b) 会消耗大量的cpu和io及内存资源

c) 推荐在程序中获取随机值

12. 禁止在where从句中对列进行函数转换和计算 ,会导致无法使用相关列上的索引

1. where date(create\_time)=’20170901’ 写成 where create\_time >= ‘20170901’ and create\_time < ‘20170902’

13. 在明显不会有重复值时使用union all而不是union

a) union 会把所有数据放在临时表中后再进行去重操作，会多消耗内存，IO，网络资源

b) union all 不会再对结果集进行去重操作

14. 拆分复杂的大sql为多个小sql

a) 目前mysql中一个sql只能使用一个cpu计算，不支持多cpu并行计算

b) sql拆分后可以通过并行执行来提高处理效率

##### 数据库操作行为规范

主要面向手动操作数据库的行为

1. 超过100万的批量写操作，要分批多次进行操作

a). 主从复制中：大批量操作可能会造成严重的主从延迟，因为当主库执行完成后，才会在从库执行

b). binlog日志为row格式时会产生大量的日志

c). 避免产生大量事务，产生阻塞，占满可用连接

2. 对大表数据结构的修改一定要谨慎

a). 可能会造成严重的锁表操作，尤其是生产环境，是不能忍受的

b). 对于大表使用pt-online-schema-change修改表结构：

c). 首先会建立一个与原表结构相同的新表

d). 然后在新表上进行表结构的修改

e). 然后把原表中的数据复制到新表中，并且增加一些触发器，以便把原表中即时新增的数据也复制到新表中

f). 在行的所有数据复制完成之后，会在原表上增加一个很准的时间锁，同时把新表命名为原表，把原表删掉

g). [实际上是把一个原子的DDL操作分解成多批次进行]

h). [避免大表修改产生的主从延迟问题]

i). [避免在对表字段进行修改时进行锁表]

3. 禁止为程序使用的账号赋予super权限

a). 当数据库连接数达到最大限制时，允许1个有super权限的用户连接

b). super权限只能留给DBA处理问题的账号使用

4. 对于程序连接数据库账号，遵循权限最小原则

a). 程序使用的数据库账号只能在一个DB下使用，不准跨库

b). 程序使用的账号原则上不准有drop权限

5. 禁止使用非本人账户登录

#### 建模工具powerdesigner

pd是一个建模工具，提供直观而便捷的交互环境，支持软件开发生命周期所有阶段的模型设计工作，包括业务流程建模、应用程序建模以及数据建模等，还提供模型报告功能，通过个阶段的设计文档可以让系统分析人员、软件开发人员、数据库管理人员以及用户之间进行有效的沟通，增强团队协作，提高软件质量。

##### pd中的模型类型

1. 需求模型（RQM）

这是一种文档式模型，它通过准确恰当地列出、解释开发过程中需要实现的功能行为来描述待开发项目。你可以为开发过程中需要使用到的各种结构化技术文档（功能或技术规格说明书，测试计划）而使用需求模型。

1. 业务流程模型（BPM）

业务流程模型主要用来描述业务功能的流程定义，是从用户角度对业务规则进行描述的一种模型。业务流程模型使用图形符号表示处理、流、消息、协作以及他们之间的相互关系。它有一个或多个起点和终点。

1. 概念数据模型（CDM）

概念数据模型主要用来描述现实世界的概念化结构，是对需求进行综合、归纳和抽象之后，形成的一个独立于具体数据库管理软件的模型。概念书模型的设计以实体-联系（E-R）模型为基础，按用户的观点对系统所需数据建模。它能够让数据库设计人员的初始阶段百通计算机系统和DBMS的具体技术问题，集中精力分析数据及其相互关系等。目标是同意业务概念，作为业务人员和技术人员之间沟通的桥梁。

1. 逻辑数据模型（LDM）

逻辑数据模型是对概念数据模型的进一步分解和细化，是具体的DBMS所支持的数据模型，如网状数据模型（Network Data Model）、层次数据模型（hierarchical Data Model）、关系数据模型（Relation Data Model）等等。逻辑数据模型是根据业务规则确定的关于业务对象、业务对象数据项、业务对象之间关系的基本蓝图。逻辑数据模型即要面向用户，又要面向系统。

逻辑数据模型的目标是尽可能详细的描述数据，但并不考虑在物理上如何实现。逻辑数据模型的设计不仅影响数据库设计的方向，还间接影响最终数据库的性能。

1. 物理数据模型（PDM）

物理数据模型用于描述数据在存储介质上的组织结构，与具体的DBMS相关。它是在逻辑数据模型的基础上，考虑各种具体的技术实现因素，进行数据库体系结构设计，真正实现数据在数据库中的表示。物理数据模型目标是为一个给定的逻辑模型选取一个最适合应用要求的物理结构。

1. 自由模型（FEM）

自由模型能够为任何类型的对象或系统提供一个上下文环境，允许自定义概念和图形符号。

1. 企业架构模型（EAM）

企业架构模型是指使用适当的方式从一个或者多个角度对一个企业的体系结构进行描述，从而产生一系列能代表企业实际情况的模型。如今，企业架构已经成为许多大公司用于理解和表述企业纤细基础设施的一个直观模型，为企业现在以及未来的信息基础设施建设提供了蓝图以及架构。

1. 数据移动模型（DMM）

数据移动模型主要用于描述模型之间的数据流动关系，利用数据移动模型可以分析和记录数据源、数据移动路径以及数据转换方式。另外，通过数据移动模型还可以完成数据库对象的复制处理以及表达数据抽取、转换和加载的过程即ETL

1. 面向对象模型（OOM）

面向对象模型采用统一建模语言（UML）描述系统的功能、结构等特性。可以从面向对象模型生成java，c++等代码，也可以逆向工程从java等文件生成面向对象模型。

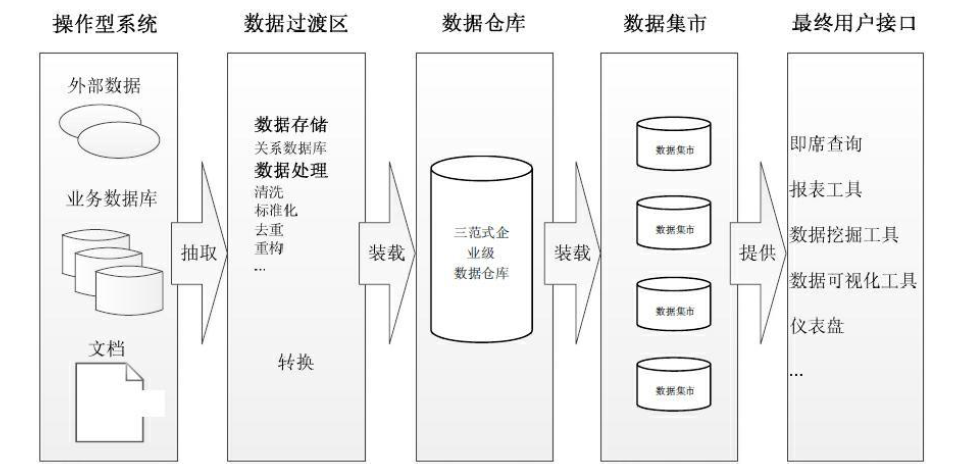
1. XML模型

XML即可扩展标记语言，是一种简单的数据存储语言。使用一系列简单的标记描述数据，而这些标记可以用方便的方式建立。

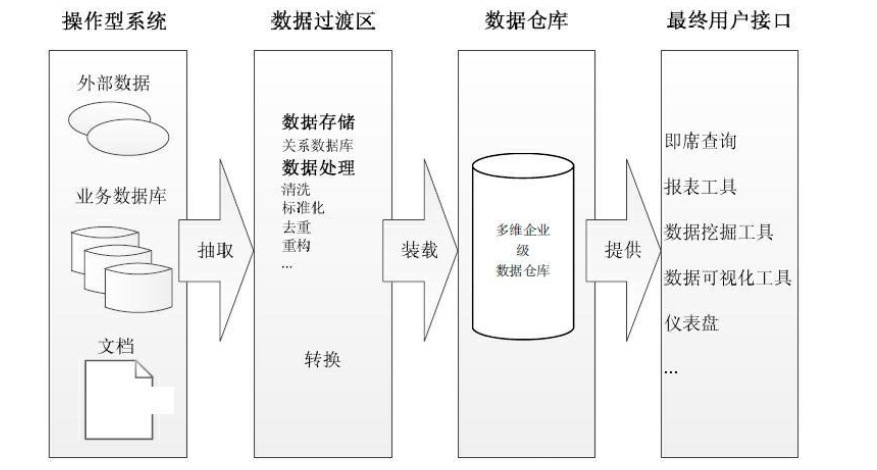
## 数据仓库

是一个面向主题的、集成的、非易失的和时变的数据集合，用于支持管理者的决策过程

### inmon架构



### kimball架构



inmon架构和kimball架构的区别就是inmon的数据仓库是三范式企业级数据仓库，kimball的数据库时多维企业级数据仓库。

### 多维数据模型及建模过程

维度模型通常以一种被称为星型模式的方式构建，一般使用下面的过程构建维度模型：

* 选择业务流程
* 声明粒度
* 确认维度
* 确认事实

1．选择业务流程

确认哪些业务处理流程是数据仓库应该覆盖的，是维度方法的基础。因此，建模的第一个步骤是描述需要建模的业务流程。例如，需要了解和分析一个零售店的销售情况，那么与该零售店销售相关的所有业务流程都是需要关注的。为了描述业务流程，可以简单地使用纯文本将相关内容记录下来，或者使用“业务流程建模标

注”（BPMN）方法，也可以使用统一建模语言（UML）或其他类似的方法。

2．声明粒度

确定了业务流程后，下一步是声明维度模型的粒度。这里的粒度用于确定事实中表示的是什么，例如，一个零售店的顾客在购物小票上的一个购买条目。在选择维度和事实前必须声明粒度，因为每个候选维度或事实必须与定义的粒度保持一致。在一个事实所对应的所有维度设计中强制实行粒度一致性是保证数据仓库应用性能和易用性的关键。从给定的业务流程获取数据时，原始粒度是最低级别的粒度。建议从原始粒度数据开始设计，因为原始记录能够满足无法预期的用户查询。汇总后的数据粒度对优化查询性能很重要，但这样的粒度往往不能满足对细节数据的查询需求。不同的事实可以有不同的粒度，但同一事实中不要混用多种不同的粒度。维度模型建立完成之后，还有可能因为获取了新的信息，而回到这步修改粒度级别。

3．确认维度

设计过程的第三步是确认模型的维度。维度的粒度必须和第二步所声明的粒度一致。维度表是事实表的基础，也说明了事实表的数据是从哪里采集来的。典型的维度都是名词，如日期、商店、库存等。维度表存储了某一维度的所有相关数据，例如，日期维度应该包括年、季度、月、周、日等数据。

4．确认事实

确认维度后，下一步也是维度模型四步设计法的最后一步，就是确认事实。这一步识别数字化的度量，构成事实表的记录。它是和系统的业务用户密切相关的，因为用户正是通过对事实表的访问获取数据仓库存储的数据。大部分事实表的度量都是数字类型的，可累加，可计算，如成本、数量、金额等。

### Data Vault模型及建模过程

Data Vault是面向细节的，可追踪历史的，一组有连接关系的规范化的表的集合。这些表可以支持一个或多个业务功能。它是一种综合了第三范式（3NF）和星型模型

优点的建模方法。其设计理念是要满足企业对灵活性、可扩展性、一致性和对需求的适应性要求，是一种专为企业级数据仓库量身定制的建模方式。它的建模步骤如下：

1．设计中心表

首先要确定企业数据仓库要涵盖的业务范围；其次要将业务范围划分为若干原子业务实体，比如客户、产品等；然后，从各个业务实体中抽象出能够唯一标识该实体的业务主键，该业务主键要在整个业务的生命周期内不会发生变化；最后，由该业务主键生成中心表。

2．设计链接表

链接表体现了中心表之间的业务关联。设计链接表，首先要熟悉各个中心表代表的业务实体之间的业务关系，可能是两个或者多个中心表之间的关系。根据业务需求，这种关系可以是1对1、1对多，或者多对多的。然后，从相互之间有业务关系的中心表中，提取出代表各自业务实体的中心表主键，这些主键将被加入到链接表中，组合构成该链接表的主键。同样出于技术的原因，需要增加代理键。

在生成链接表的同时，要注意如果中心表之间有业务交易数据的话，就需要在链接表中保存交易数据，有两种方法，一是采用加权链接表，二是给链接表加上附属表来处理交易数据。

3．设计附属表

附属表包含了各个业务实体与业务关联的详细的上下文描述信息。设计附属表，首先要收集各个业务实体在提取业务主键后的其他信息，比如客户住址、产品价格等；由于同一业务实体的各个描述信息不具有稳定性，会经常发生变化，所以，在必要的时候，需要将变化频率不同的信息分隔开来，为一个中心表建立几个附属表，然后提取出该中心表的主键，作为描述该中心表的附属表的主键。当业务实体之间存在交易数据的时候，需要为没有加权的链接表设计附属表，也可以根据交易数据的不同变化情况设计多个附属表。

4．设计必要的PIT表

Point—In—Time表是由附属表派生而来的。如果一个中心表或者链接表设计有多个附属表的话，而为了访问数据方便，就有用到PIT表的可能。PIT表的主键也是由其所归属的中心表提取而来，该中心表有几个附属表，PIT表就至少应该有几个字段来存放各个附属表的变化对比时间。

建立Data Vault模型时应该参照如下的原则：

（1）关于中心表的原则

中心表的主键不能够直接“伸入”到其他中心表里面。就是说，不存在父子关系的中心表。各个中心表之间的关系是平等的，这也正是Data Vault模型灵活性与扩展性

之所在。

中心表之间必须通过链接表相关联，通过链接表可以连接两个以上的中心表。

必须至少有两个中心表才能产生一个有意义的链接表。

中心表的主键总是“伸出去”的（到链接表或者附属表）。

（2）关于链接表的原则

链接表可以跟其他链接表相连。

中心表和链接表都可以使用代理键。

业务主键从来不会改变，就是说中心表的主键也即链接表的外键不会改变。

（3）关于附属表的原则

附属表必须是连接到中心表或者链接表上才会有确定的含义。

附属表总是包含装载时间和失效时间，从而包含历史数据，并且没有重复的数据。

由于数据信息的类型或者变化频率快慢的差别，描述信息的数据可能会被分隔到多个附属表中去。

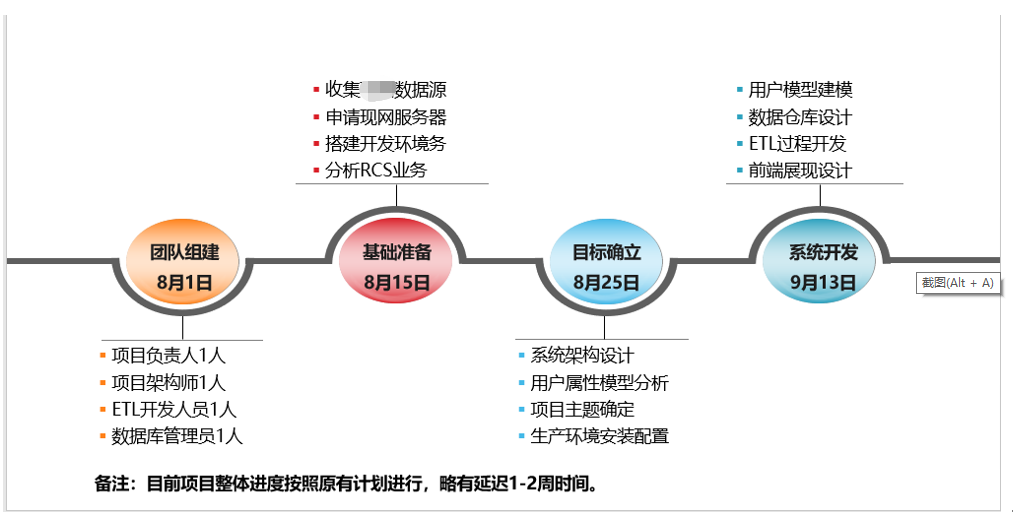
### 数仓开发流程

#### 需求分析

参见《数据仓库项目需求分析.doc》

#### 开发计划

设置进度节点



制定项目计划



更详细规划《数据仓库项目计划.xls》

#### 概念模型设计

  按用户的观点来对数据和信息建模，就是概念模型，是现实世界中数据特征的抽象。它能够能够比较真实地模拟现实世界,容易被人理解和便于计算机实现。它以实体－联系(Entity-RelationShip,简称E-R)理论为基础，并对这一理论进行了扩充。它从用户的观点出发对信息进行建模，主要用于[数据库](http://lib.csdn.net/base/mysql" \t "_blank" \o "MySQL知识库)的概念级设计。

概念模型就是在了解了用户的需求，用户的业务领域工作情况以后，经过分析和总结，提炼出来的用以描述用户业务需求的一些概念的东西。如销售业务中的“客户”和“定单”，还有就是“商品”，“业务员”。

通常人们先将现实世界抽象为概念世界，然后再将概念世界转为机器世界。换句话说，就是先将现实世界中的客观对象抽象为实体(Entity)和联系 (Relationship),它并不依赖于具体的计算机系统或某个DBMS系统，这种模型就是我们所说的CDM;然后再将CDM转换为计算机上某个 DBMS所支持的数据模型，这样的模型就是物理数据模型,即PDM。

概念模型的设计步骤：

1. 表示方法

E-R图主要是由实体、属性和联系三个要素构成的。在E-R图中，使用了下面四种基本的图形符号。

1. 确定系统实体、属性及联系

系统分析阶段建立数据字典和数据流程图->建立概念模型->逻辑模型->物理模型。利用系统分析阶段建立的数据字典，并对照数据流程图对系统中的各个数据项进行分类、组织，确定系统中的实体、实体的属性、标识实体的码以及实体之间联系的类型。

在数据字典中“数据项”是基本数据单位，一般可以作为实体的属性。“数据结构”、“数据存储”和“数据流”条目都可以作为实体，因为它们总是包含了若干的数据项。作为属性必须是不可再分的数据项，也就是说在属性中不能包含其他的属性。

1. 确定局部（分）E-R图

根据上面的分析，可以画出部分实体－联系图。

在这些实体中有下画线的属性可以作为实体的码，这几个实体之间存在着1：1、l：n和m：n几种联系。

1. 集成完整（总）E-R图

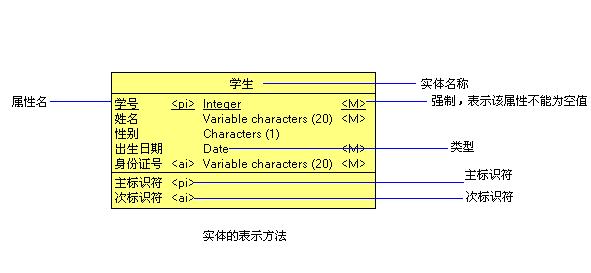
各个局部（分）E-R图画好以后，应当将它们合并起来集成为完整（总）E-R图。在集成时应当注意如下几点：

（1）消除不必要的冗余实体、属性和联系。

（2）解决各分E-R图之间的冲突。

（3）根据情况修改或重构E-R图。

一个概念模型的学生实体如下图



概念模型是逻辑设计的基础。概念数据模型的内容包括重要的实体及实体之间的关系。在概念数据模型中不包括实体的属性，也不用定义实体的主键。这是概念数据模型和逻辑数据模型的主要区别。概念数据模型的目标是统一业务概念，作为业务人员和技术人员之间沟通的桥梁，确定不同实体之间的最高层次的关系。在有些数据模型的设计过程中，概念数据模型是和逻辑数据模型合在一起进行设计的。

#### 逻辑设计

逻辑模型就是要将概念模型具体化。要实现概念模型所描述的东西，需要那些具体的功能和处理那些具体的信息。这就到了需求分析的细化阶段。还以销售业务为例：“客户”信息基本上要包括：单位名称，联系人，联系电话，地址等属性；“商品”信息基本上要包括：名称，类型，规格，单价等属性；“定单”信息基本上要包括：日期和时间属性。并且“定单”要与“客户”，“业务员”和“商品”明细关联。系统需要建立几个数据表：业务员信息表，客户信息表，商品信息表，定单表。系统要包括几个功能：业务员信息维护，客户信息维护，商品信息维护，建立销售定单。这些均属于建立逻辑模型，这些说明只表明系统要实现什么，但怎样实现，用什么工具实现还没有讲，后者属于物理模型范围。

逻辑数据模型反映的是系统分析设计人员对数据存储的观点，是对概念数据模型进一步的分解和细化。逻辑数据模型是根据业务规则确定的，关于业务对象、业务对象的数据项及业务对象之间关系的基本蓝图。 逻辑数据模型的内容包括所有的实体和关系，确定每个实体的属性，定义每个实体的主键，指定实体的外键，需要进行范式化处理。逻辑数据模型的目标是尽可能详细的描述数据，但并不考虑数据在物理上如何来实现。逻辑数据建模不仅会影响数据库设计的方向，还间接影响最终数据库的性能和管理。如果在实现逻辑数据模型时投入得足够多，那么在物理数据模型设计时就可以有许多可供选择的方法。

逻辑结构设计的任务，就是把概念结构设计阶段建立的基本E-R图，按选定的管理系统软件支持的数据模型（层次、网状、关系），转换成相应的逻辑模型。这种转换要符合关系数据模型的原则。

E-R图向关系模型的转换是要解决如何将实体和实体间的联系转换为关系，并确定这些关系的属性和码。这种转换一般按下面的原则进行：

（1）一个实体转换为一个关系，实体的属性就是关系的属性，实体的码就是关系的码。

（2）一个联系也转换为一个关系，联系的属性及联系所连接的实体的码都转换为关系的属性，但是关系的码会根据联系的类型变化，如果是：

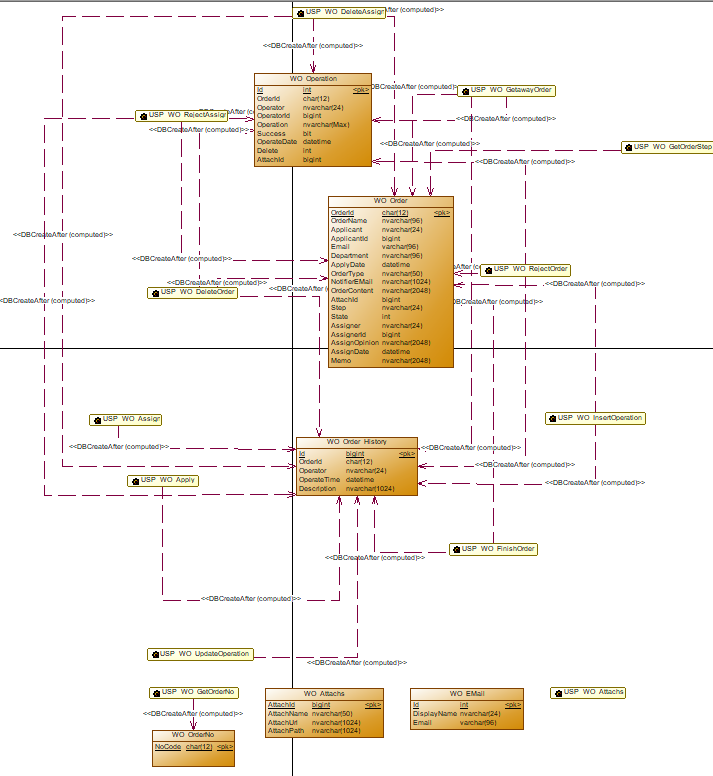
1：1联系，两端实体的码都成为关系的候选码。

1：n联系，n端实体的码成为关系的码。

m：n联系，两端实体码的组合成为关系的码。

#### 物理设计

物理模型就是针对上述逻辑模型所说的内容，再具体的物理介质上实现出来，也就是系统需要哪些表，表中有哪些字段，字段的类型是什么，表之间的关系是一对一还是一对多还是多对多。物理设计一般要具体到使用哪种数据库管理软件。物理模型就可以直接生成数据库的创建脚本，以及数据库设计文档。一个物理模型如下图所示



#### 详细设计

见《数据仓库系统设计文档.doc》和《ETL设计说明书.doc》

#### 项目管理

很多公司在人员分工，任务分派等使用邮件，但也有些规模比较大的公司是用项目管理软件，jira就是一个比较常用的项目与事务跟踪工具，被广泛应用于缺陷跟踪、客户服务、需求收集、流程审批、任务跟踪、项目跟踪和敏捷管理等工作领域。它的主要功能有：

问题追踪和管理：用它管理项目，跟踪任务、bug、需求，通过jira的邮件通知功能进行协作通知，在实际工作中使工作效率提高很多

问题跟进情况的分析报告：可以随时了解问题和项目的进展情况

项目类别管理功能：可以将相关的项目分组管理

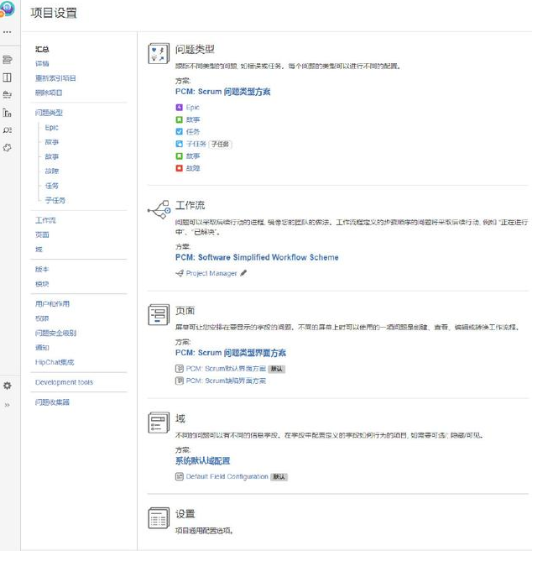
组件/模块负责人功能：可以将项目的不同组件/模块指派相应的负责人，来处理所负责的组件的Issues

项目email地址功能：每个项目可以有不同的email（该项目的通知邮件从该地址发出）

无限制的工作流：可以创建多个工作流为不同的项目使用

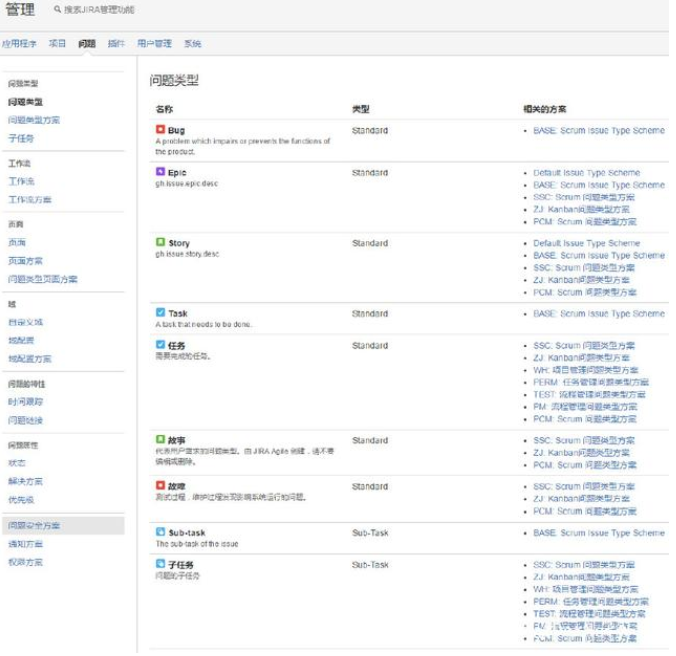
创建项目





创建任务和问题（issue

）



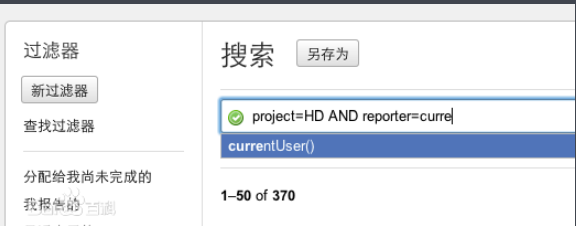
设置人员权限和角色



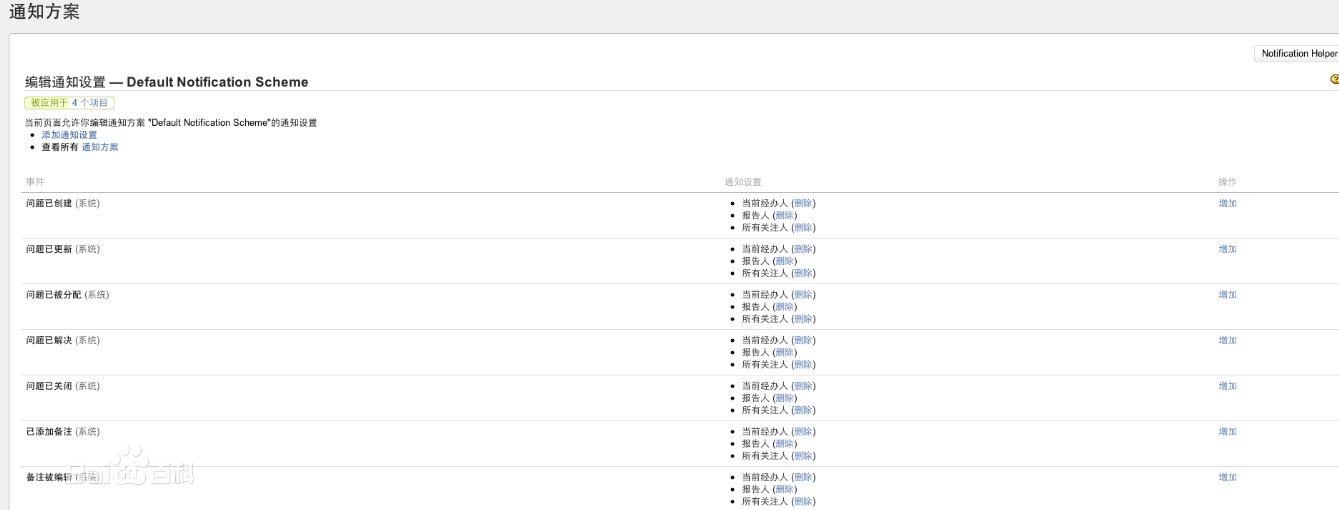
对工作任务进行汇总



搜索特定的任务



问题状态更改，给相关人员发送邮件



你需要做的工作内容清晰明了



#### 部署实施(oozie, azkaban~>任务调度的框架)

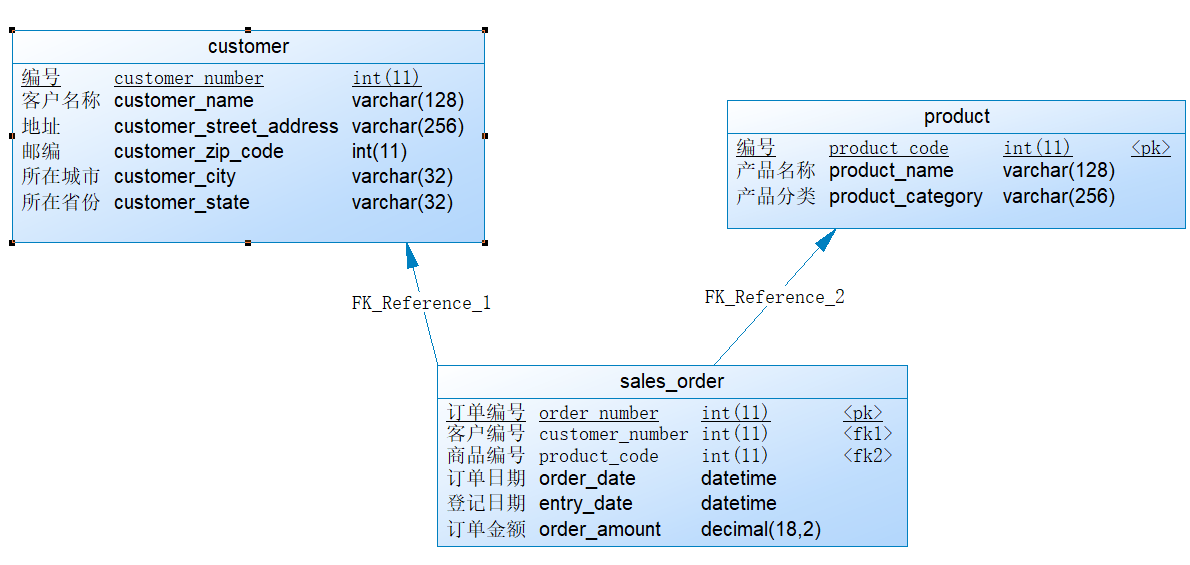
见《数据仓库建设项目实施方案建议书.doc》

## 完整案例

### 准备环境

#### 业务库

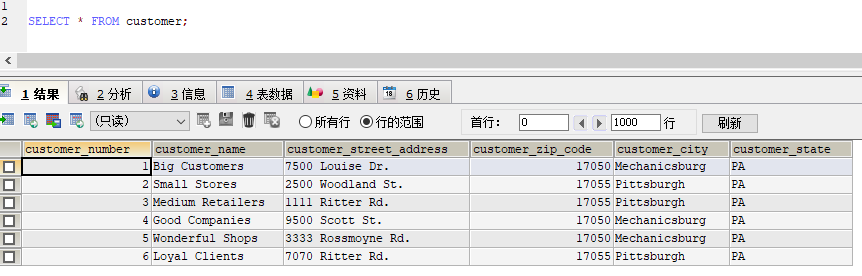
源系统是mysql库，数据模型如下

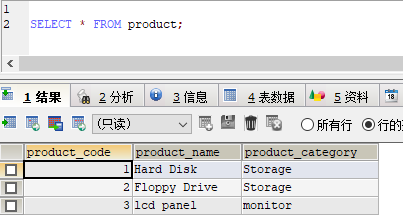


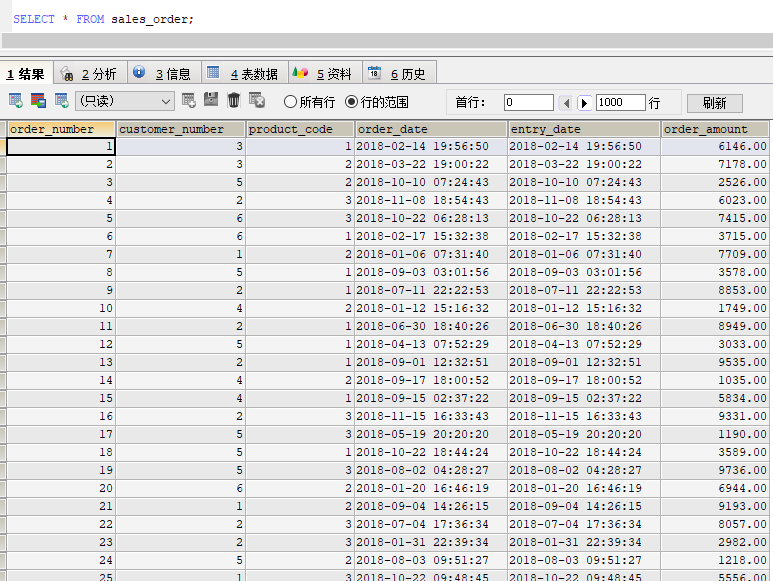
建表语句如下：

|  |
| --- |
| /\*==============================================================\*/  /\* DBMS name: MySQL 5.0 \*/  /\* Created on: 2018/11/23 1:09:10 \*/  /\*==============================================================\*/  CREATE DATABASE IF NOT EXISTS sales\_source DEFAULT CHARSET utf8 COLLATE utf8\_general\_ci;  USE sales\_source;  DROP TABLE IF EXISTS customer;  DROP TABLE IF EXISTS product;  DROP TABLE IF EXISTS sales\_order;  /\*==============================================================\*/  /\* Table: customer \*/  /\*==============================================================\*/  CREATE TABLE customer  (  customer\_number INT(11) NOT NULL AUTO\_INCREMENT,  customer\_name VARCHAR(128) NOT NULL,  customer\_street\_address VARCHAR(256) NOT NULL,  customer\_zip\_code INT(11) NOT NULL,  customer\_city VARCHAR(32) NOT NULL,  customer\_state VARCHAR(32) NOT NULL,  PRIMARY KEY (customer\_number)  );sales\_source  /\*==============================================================\*/  /\* Table: product \*/  /\*==============================================================\*/  CREATE TABLE product  (  product\_code INT(11) NOT NULL AUTO\_INCREMENT,  product\_name VARCHAR(128) NOT NULL,  product\_category VARCHAR(256) NOT NULL,  PRIMARY KEY (product\_code)  );  /\*==============================================================\*/  /\* Table: sales\_order \*/  /\*==============================================================\*/  CREATE TABLE sales\_order  (  order\_number INT(11) NOT NULL AUTO\_INCREMENT,  customer\_number INT(11) NOT NULL,  product\_code INT(11) NOT NULL,  order\_date DATETIME NOT NULL,  entry\_date DATETIME NOT NULL,  order\_amount DECIMAL(18,2) NOT NULL,  PRIMARY KEY (order\_number)  );  /\*==============================================================\*/  /\* insert data \*/  /\*==============================================================\*/  INSERT INTO customer  ( customer\_name  , customer\_street\_address  , customer\_zip\_code  , customer\_city  , customer\_state  )  VALUES  ('Big Customers', '7500 Louise Dr.', '17050',  'Mechanicsburg', 'PA')  , ( 'Small Stores', '2500 Woodland St.', '17055',  'Pittsburgh', 'PA')  , ('Medium Retailers', '1111 Ritter Rd.', '17055',  'Pittsburgh', 'PA'  )  , ('Good Companies', '9500 Scott St.', '17050',  'Mechanicsburg', 'PA')  , ('Wonderful Shops', '3333 Rossmoyne Rd.', '17050',  'Mechanicsburg', 'PA')  , ('Loyal Clients', '7070 Ritter Rd.', '17055',  'Pittsburgh', 'PA')      INSERT INTO product(product\_name,product\_category) VALUES  ('Hard Disk','Storage'),  ('Floppy Drive','Storage'),  ('lcd panel','monitor')  DROP PROCEDURE IF EXISTS usp\_generate\_order\_data;  DELIMITER //  CREATE PROCEDURE usp\_generate\_order\_data()  BEGIN  DROP TABLE IF EXISTS tmp\_sales\_order;  CREATE TABLE tmp\_sales\_order AS SELECT \* FROM sales\_order WHERE 1=0;  SET @start\_date := UNIX\_TIMESTAMP('2018-1-1');  SET @end\_date := UNIX\_TIMESTAMP('2018-11-23');  SET @i := 1;  WHILE @i<=100000 DO  SET @customer\_number := FLOOR(1+RAND()\*6);  SET @product\_code := FLOOR(1+RAND()\* 3);  SET @order\_date := FROM\_UNIXTIME(@start\_date+RAND()\*(@end\_date-@start\_date));  SET @amount := FLOOR(1000+RAND()\*9000);  INSERT INTO tmp\_sales\_order VALUES (@i,@customer\_number,@product\_code,@order\_date,@order\_date,@amount);  SET @i := @i +1;  END WHILE;  TRUNCATE TABLE sales\_order;  INSERT INTO sales\_order  SELECT NULL,customer\_number,product\_code,order\_date,entry\_date,order\_amount  FROM tmp\_sales\_order;  COMMIT;  DROP TABLE tmp\_sales\_order;  END //  CALL usp\_generate\_order\_data(); |

建完库后的表和数据如下







#### 数据仓库

数仓是建立在hive上，有两层（ODS层rds库）和DW层（tds库），存储格式日期维度textfile，其他orc。

##### 模型搭建

星型模式是维度模型最简单的形式，也是比较常用的模型，我们的案例采用星型模型。所谓星型模型就是以一个事实表为中心，周围围绕多个维度表。对维度表做进一步规范化后形成的模型叫雪花模型，含有很多维度表的星型模型有时被称为蜈蚣模型，蜈蚣模型的维表往往只有很少的几个属性，这样可以简化维度表的维护，但同时查询数据的时候会有很多的表连接，严重时会使模型难以使用，因此要尽量避免这种模型。

星型模型将业务分为事实和维度。

事实是业务数据的度量值，比如销售额、销售数量等，它记录了特定事件的量化指标，一般是度量值和指向维表的外键组成。事实表的粒度级别通常会设计的比较低,事实表有三种类型:

事务事实表:最低粒度级别的事实表,记录原始的操作型事件.

快照事实表:记录给定时间点的事实,如月底账户余额

累积事实表:记录给定时间点的聚合事实,如当月的销售金额.

维度是对事实数据属性的描述，如日期，省份，地区等,维度表的数据量通常不大,常用的维度表有:

时间维度表,每个数据仓库都需要一个时间维度表。

地理维度表:描述位置信息的数据,如国家,省份,城市,区县,邮编等

产品维度表:描述产品及其属性

人员维度表:描述人员相关信息,部门员工表等

范围维度表:描述分段数据的信息等,比如信用等级

代理键:一般事实表和维表都有主键,单仍会设置一个代理键,所谓代理键说白了及时业务无关的自增主键,因为维表的主键有可能会产生变化,即变化维.

星型模型是非规范化的,不受关系数据库的范式规则的约束,当所有的维度进行规范化后也叫做”雪花化”,就是雪花模型了,具体的做法是将低基数(维表中的行数少,比如性别)的属性从维度表中移除并形成单独的维表,维表就具有了层次关系(父子),减少了维表数据的冗余,因此大数据量下雪花比星型节省空间,但是相对的查询要关联的表多,因此也就变的复杂.有些设计底层使用雪花模型,上层用表连接简历视图模拟星型模型,这种方式通过对维度的规范化节省了空间,同时又对用户屏蔽了查询的复杂性,但是视图对于查询效率的提升相对于联合查询来说并没有得到提升,对开发效率有提升,性能有损失

###### 选择业务流程

维度方法的基础是首先确认哪些业务处理流程是数据库需要覆盖的,因此建模的第一个步骤是描述需要建模的业务流程,描述业务流程,可以简单的使用文本记录下俩或者使用MPMN(业务流程建模标注)的方法,也可以使用UML等.

我们的案例业务很明确就是:销售订单

###### 粒度，

粒度用于确定事实表中表示的是什么,在选择

事实表存储最细粒度的事务记录，每小时更新增量，凌晨2两点更新昨天全量

###### 确认维度：

产品、客户以及日期，日期维度用于业务集成，每个数据仓库都应该有一个日期维度，日期维度数据一旦生成就不会改变，因此不需要版本号、生效日期和过期日期，一般情况下直接生成10年或者20年的数据，初始化的数据远高于数据仓库的有效时长即可。

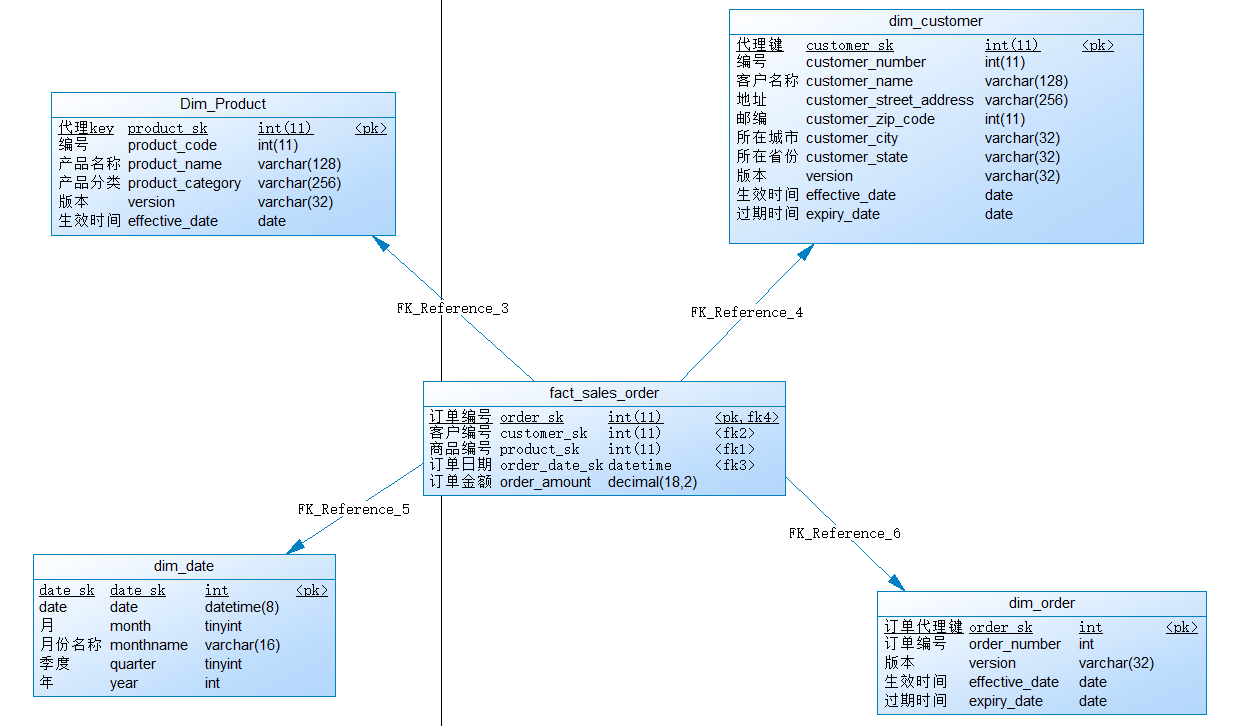
在有变化的维度表上增加版本号、生效日期、过期日期，能看到维度的历史变化，当维度属性发生变化的时候，根据不同的策略，生成一条新的维度记录或者更改原记录。渐变维 slow changing dimensions SCD

代理键是维度表的主键，一般加sk表示即surrogate key,是每行记录的唯一标识,由系统生成的主键,不是应用数据,没有业务含义.

###### 确认事实：

订单是唯一事实，订单金额是唯一度量，按天分区。

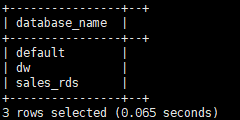
###### 建立物理模型



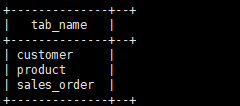
###### 建库、装载数据

|  |
| --- |
| 连接hive  创建rds库 create database sales\_rds;  创建表  /\*==============================================================\*/  /\* Table: customer \*/  /\*==============================================================\*/  CREATE TABLE sales\_rds.customer  (  customer\_number INT ,  customer\_name VARCHAR(128) ,  customer\_street\_address VARCHAR(256) ,  customer\_zip\_code INT ,  customer\_city VARCHAR(32) ,  customer\_state VARCHAR(32)  );  /\*==============================================================\*/  /\* Table: product \*/  /\*==============================================================\*/  CREATE TABLE sales\_rds.product  (  product\_code INT,  product\_name VARCHAR(128) ,  product\_category VARCHAR(256)  );  /\*==============================================================\*/  /\* Table: sales\_order \*/  /\*==============================================================\*/  CREATE TABLE sales\_rds.sales\_order  (  order\_number INT ,  customer\_number INT,  product\_code INT ,  order\_date timestamp ,  entry\_date timestamp ,  order\_amount DECIMAL(18,2)  )  row format delimited fields terminated by '\t'  创建库dw create database dw;  创建表  create table dim\_Product  (  product\_sk int ,  product\_code int ,  product\_name varchar(128),  product\_category varchar(256),  version varchar(32),  effective\_date date,  expiry\_date date  )  clustered by (product\_sk ) into 8 buckets  stored as orc tblproperties('transactional'='true');  /\*==============================================================\*/  /\* Table: dim\_customer \*/  /\*==============================================================\*/  create table dim\_customer  (  customer\_sk int ,  customer\_number int ,  customer\_name varchar(128),  customer\_street\_address varchar(256),  customer\_zip\_code int,  customer\_city varchar(32),  customer\_state varchar(32),  version varchar(32),  effective\_date date,  expiry\_date date  )  clustered by (customer\_sk ) into 8 buckets  stored as orc tblproperties('transactional'='true');  /\*==============================================================\*/  /\* Table: dim\_date \*/  /\*==============================================================\*/  create table dw.dim\_date  (  date\_sk int ,  date date,  month tinyint,  month\_name varchar(16),  quarter tinyint,  year int  ) row format delimited fields terminated by ','  stored as textfile;  /\*==============================================================\*/  /\* Table: dim\_order \*/  /\*==============================================================\*/  create table dim\_order  (  order\_sk int ,  order\_number int,  version varchar(32),  effective\_date date,  expiry\_date date  )  clustered by (order\_sk ) into 8 buckets  stored as orc tblproperties('transactional'='true');  ;  /\*==============================================================\*/  /\* Table: fact\_sales\_order \*/  /\*==============================================================\*/  create table fact\_sales\_order  (  order\_sk int ,  customer\_sk int ,  product\_sk int ,  order\_date\_sk int ,  order\_amount decimal(18,2)  )  partitioned by (order\_date string)  clustered by (order\_sk ) into 8 buckets  row format delimited fields terminated by '\t'  stored as orc tblproperties('transactional'='true'); |

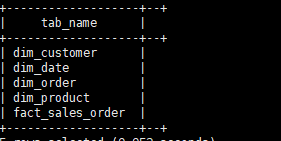
创建完表库如下



sales\_rds表如下



dw的表如下



注意,dw库的表除dim\_date存储格式是textfile外,其他的表均为orc,并且设置表属性支持事务

虽然表在模型设计中设置了主键,但是在hive中没有主键,外键唯一性约束,非空约束这些概念

###### 数据仓库初始化 装载日期维度表

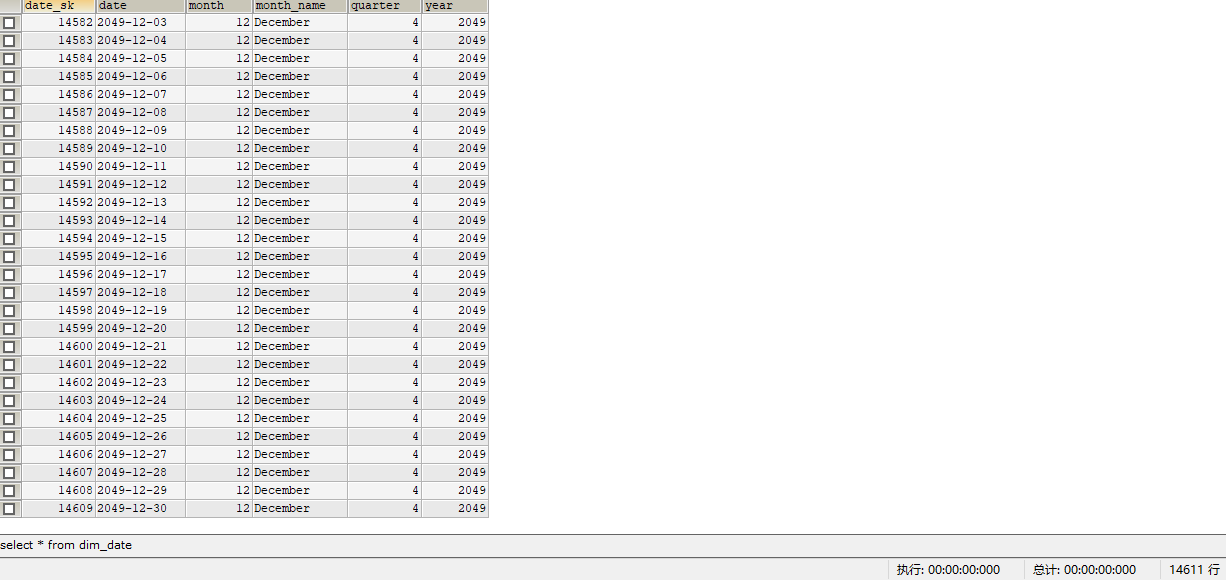
日期维度是数据仓库中一个很重要的角色,一般来说每个数据仓库都要有日期维度,数据仓库存储的是历史数据,分析的中一个很重要的指标是基于时间的一个业务的变化趋势,因此,数据仓库中都包含这个维度,装载(load)时间维度表有三种常用方式:预装载,每天装载一天,从源数据装载日期.这三种方式中预装载是最常见的,使用这个方式,在数据仓库的生命周期中,只需要装载一次,一般开始时间选做源数据的最小时间,结束时间选做一个数据仓库的有效时间,比如50年(如果觉得不保险可以选100年以后),这种方式的缺点是提早消耗磁盘的空间,可能不需要所有的日期也就是稀疏使用,这两个问题基本上可以忽略,磁盘占空间也很有限。

* 从源数据装载日期，源数据库中可能就存在着这样一个表，同步到数据仓库就可以。

我们的例子中，加入在sales\_source中存在一个dim\_date,我们建一个存储过程来实现数据的生成,脚本如下

|  |
| --- |
| DELIMITER //  CREATE PROCEDURE USP\_Load\_Dim\_Date(dt\_start DATE,dt\_end DATE)  BEGIN  WHILE dt\_start<=dt\_end DO  INSERT INTO dim\_date (`date`,`month`,`month\_name`,`quarter`,`year`)  VALUES (dt\_start,MONTH(dt\_start),MONTHNAME(dt\_start),QUARTER(dt\_start),YEAR(dt\_start));  SET dt\_start =ADDDATE(dt\_start,1);  END WHILE;  COMMIT;  END //  CALL USP\_Load\_Dim\_Date('2010-1-1','2050-1-1'); |

查看一下数据 SELECT \* FROM dim\_date



直接为hive数仓生成日期表,我们需要写一个shell脚本,生成日期数据文件,上传到hdfs对应的日期维度表的目录即可,注意如果是在windows下编辑的脚本上传到linux下,执行脚本的时候会报错,windows下每行都会添加\r\n,在linux下vim编辑一下文件,使用命令查看:set ff? 看是什么格式的,如果是dos,重新设置一下格式:set ff=unix,再保存退出即可,如果脚本出错一般会提示是第几行,可以使用:set number 或者:set nu来显示行号,以便查找错误

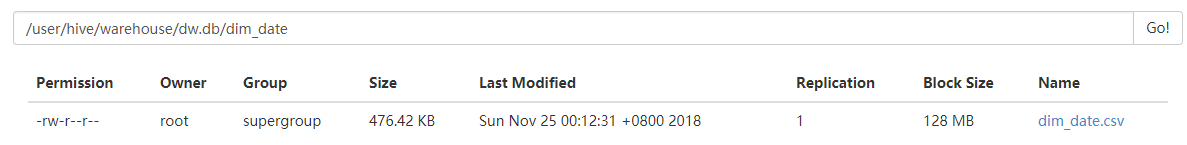
脚本如下:

|  |
| --- |
| #!/bin/bash  date1="$1"  date2="$2"  tempdate=`date -d "$date1" +%F`  tempdateSec=`date -d "$date2" +%s`  enddateSec=`date -d "$date2" +$s`  min=1  #max=`expr \( $enddateSec - $tempdateSec \) / \( 24 \\* 60 \\* 60 \) + 1`  max=14611  cat /datas >./dim\_date.csv  while [ $min -le $max ]  do  month=`date -d "$tempdate" +%m`  month\_name=`date -d "$tempdate" +%B`  quarter=`echo $month | awk '{print int(($0-1)/3 +1 }'`  year=`date -d "$tempdate" +%Y`  echo ${min}","${tempdate}","${month}","${month\_name}","${quarter}","${year} >> ./dim\_date.csv  tempdate=`date -d "+$min day $date1" +%F`  tempdateSec=`date -d "+min day $date1" +%s`  min=`expr $min + 1`  done |

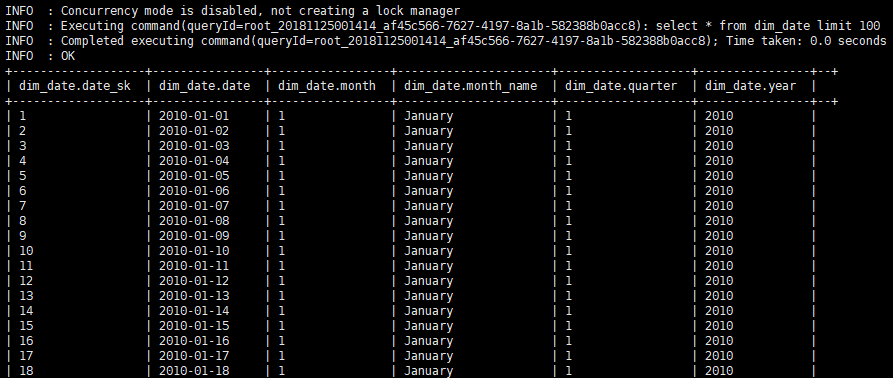
脚本保存为generate\_dim\_date.sh,执行脚本 generate\_dim\_date.sh 2010-1-1,2050-1-1

将生成的文件上传到hdfs下面/user/hive/warehose/dw.db/dim\_date/

|  |
| --- |
| hdfs dfs -put -f dim\_date.csv /user/hive/warehouse/dw.db/dim\_date |



进入hive查看一下表 select \* from dim\_date limit 100;



至此这个简单的数仓模型就搭建完成了。

##### ETL过程

###### Extract数据抽取

数据仓库搭建完成后,下一步要做的工作就是数据仓库中数据的填充,也就是我们的ETL过程,ETL可以说是数据仓库系统中最基础也是最重要的环节.我们前面讲过数据仓库的数据来源是业务系统，业务系统可能同时使用多种数据库系统，这些系统在物理上各自独立，在逻辑上有相互联系，数据仓库要从中捕获变化的数据，实现增量数据抽取。

设计ETL过程的首要步骤是建立有效的逻辑数据映射，它是建立ETL工作的指南，明确源数据组件和目标组件的对应关系，ETL工程师通过逻辑数据映射来进行ETL开发。

简单的说逻辑数据映射是指源系统中表和数据仓库中的表的对应关系，一般使用excel进行记录，包括以下组成部分：

* 源系统组件：

包括数据源名称，源表名，源列名及其数据类型，数据源名称通常指连接数据源的数据库连接字符串，包括数据库地址，实例名称，用户名，密码等。如果源数据是文件，数据源就是包括文件名的完整路径。源表是源数据的库中表名称，源数据中有很多的表，我们只需要列出与数据仓库相关的表即可。源列名是生成目标表的相关列。需要注意的是这些源表和列在最映射的时候要标注对应的含义，很多时候源数据的表和字段命名非常不规范，具体的含义就要去找负责的接口人进行整理，并进行归档，这是一个很繁琐的过程，早期的很多业务系统在软件开发的时候非常不规范，有的字段甚至是拼音缩写，很难搞清楚是什么含义，整理的过程就是数据对接的过程，一般叫做数据探查。

* 目标组件

目标组件就是数据仓库中出现的表名称，表类型（事实表、维度表、子维度表等）、列名称及数据类型和SCD类型（slow change dimesion 缓慢变化维）。为保持业务数据和数据仓库中的数据始终处于一致，就要对缓慢变化维进行处理，SCD类型有覆盖（type 1，不产生历史记录）、添加新数据（type2，保留历史信息），添加历史列（type2，维护少量变化，只追踪到上一次的历史信息），常用的就是type 1和type2。

* 转换操作

源数据到目标数据所需要进行的详细操作，一般用伪代码来编写，因为涉及到多个库或者多个表，一般是用组合的方式中间用圆点分隔:“数据库名.表名.列名”这种方式。如果映射描述是空的则表示不需要转换，直接加载即可。

逻辑数据映射不但是ETL开发工程师的开发指南，还是对开发过程中对数据的最终审查的依据。因此，必须清晰的描述转换过程中所包含的动作流程，不能有任何存疑的地方。逻辑映射文件归纳为元数据，源数据结构，目标结构，数据转换规则，映射关系，数据的上下文等，是开发过程中必须要有的文档，为ETL工程师提供指南，追踪数据来源和转换信息，也有助于设计人员理解系统环境变化所造成的影响。逻辑数据映射中还应该标识出一些需要引起重视的操作，比如隐式转换引起长度变化时，可能会丢失数据，比如utf8转换为Latin1时，字符集的变化会引起字节数量的增减，因此要在文档中标识出来，提醒开发人员注意。

建立逻辑映射的步骤

1 识别数据源，有时候业务系统会经过很多次的迭代，有很多遗留系统，标识数据源就会变的非常复杂，需要去找各个系统的接口人共同确定。

2 收集源系统的文档，从文档中找所需的列含义说明，确保数据源正确无误

3 建立源系统跟踪报告，报告应显示每个数据源的负责人、开发人员和使用者。还要包含数据源的特性：所属主题域，接口名称，业务名称，业务部门，接口人，数据库管理系统，生产服务器，数据大小，数据复杂度，每天事务数，优先级，日常使用数量，使用情况，系统平台和其他说明等。

4 建立目标数据仓库的实体关系图。实体关系统表示实体（表）之间的相互关系，通过连线表示，每个实体应有唯一标识，每个列的数据类型，实体之间的关系，这是非常重要的，会影响的数据抽取的顺序，也就是数据流转的过程。

5 建立模型映射，不同的映射类型所做的处理不一样。

* + 一对一：一个源数据的数据实体对应一个目标模型的实体。如果两者类型一致，直接映射，如果两者类型不一致，需经过转换再进行映射
  + 一对多：一个源数据的实体对应多个目标模型的实体，是一个将源实体拆分为多个目标的情况。
  + 一对零：源数据的实体没有对应的目标实体，也就是不在处理的范围内
  + 零对一：源数据的实体没有，而目标模型中有，比如典型的时间维度等
  + 多对一：多个源系统的数据实体对应一个目标模型的数据实体，这个时候取源数据就需要多表关联
  + 多对多，多个源系统的数据实体对应多个目标模型的数据实体

6 建立列的映射，这个映射的类型跟表的映射类型相同。

* + 一对一
  + 一对多
  + 一对零
  + 零对一：目标实体的列没有与数据源中的列对应起来，例如维表中的代理键，SCD的版本号，起始时间和过期时间等
  + 多对一，在抽取的数据的时候往往需要一些字符串的连接函数如concat等
  + 多对多

7 收集业务规则

数据模型有了，还必须对数据内容本身进行分析，来收集所有的业务规则，例如，是否存在非日期类型的列中存储日期值，数据是否允许为空，如果不为空默认值是什么，一般情况下业务库的设计原则是所有的列都不允许为NULL，处理NULL的业务逻辑是什么等。

抽取方式

从源数据导入到数据仓库或者贴源层有两种方式，从源数据拉取数据（pull）和请求源数据推送到数据仓库（push）。一般来讲，后一种方式需要增加业务系统的功能才能进行推送，这个在现实情况中不大行的通，一方面影响业务系统的性能，另一方面增加开发者的工作量，理论上讲，数据仓库不应该要求对源系统做任何的改造，因此一般都采用拉取数据的方式。近几年来数据部门基本上是稍有点规模的公司的标配，因此在规划新的业务系统的时候，会考虑到数据的共享问题，很多解决方案是业务系统将数据发送到kafka，需要的数据的部门去kafka中消费数据，数据仓库也有一部分是从kafka中拉取数据。

确定了数据的抽取方式，还需要确定数据的抽取类型，即全量抽取还是增量抽取。如果数据量小并且容易处理，一般采用全量抽取即可。这种方式适合基础编码类的数据源，最典型的是我们通常所说的数据字典类的码表，在数据仓库中就是维表。如果数据量很大，就只能抽取变化的源数据，即最后一次抽取以来发生了变化的数据。这种数据抽取模式成为变化数据捕获，简称CDC（change data caputer）。

CDC大体分为两种侵入式和非侵入式。侵入式是指CDC操作会给源系统带来性能影响。只要CDC对源库进行了SQL语句操作，就可以认为是侵入式的。常用的CDC方法也有以下四种

* 基于时间戳的CDC

时间戳：要求源数据里有插入时间和更新时间列，取新增数据根据插入时间，取修改数据根据更新时间

自增列：根据自增列取新增数据

这个方法最常用也最简单，简单的sql语句就可以实现

|  |
| --- |
| select \* from table where create\_time > last\_load\_time  select \* from table where update\_time>last\_load\_time  select \* from table where id> last\_load\_id |

但是缺点也很明显,不能记录删除操作,触发有逻辑删除的功能,无法识别过多次更新,只能拿到最后一次的,一般为批量操作,不具有实时性,而且具有侵入性.

* 基于触发器的CDC

当执行insert,update,delete这些SQL语句是,可以使用数据库的触发器来执行一些动作,比如触发器将变更的数据保存到临时表中,然后从临时表中抽取数据到贴源层,大多数情况下,触发器会降低业务系统的性能,因此这种方式使用的不多.作为替代方案,可以使用源数据库的复制功能,复制一份数据到备库上,在备库上建立触发器,是个很有效的方法,而且没有侵入性.这个方式需要额外的存储空间,看起来是冗余数据,实际上对业务库来讲,实现主库上进行写操作,备库上进行读操作,实现读写分离,是提高性能和高可用一个手段.

* 基于快照的CDC

如果没有时间戳也不允许使用触发器,就要使用快照表了.可以通过比较原表和快照表来获得变化数据。所谓快照就是一次性抽取源数据中的全部数据，把这些数据加载到ODS层，下次需要同步的时候，再从源数据抽取全部的数据，并把这些数据放到ODS层，比较这两个版本的数据，找出变化数据。一般通过SQL语句可以实现，比如有个表有两列：id和name，在临时过渡区有两个快照 snapshot\_1，snapshot\_2在mysql中可以这么写：

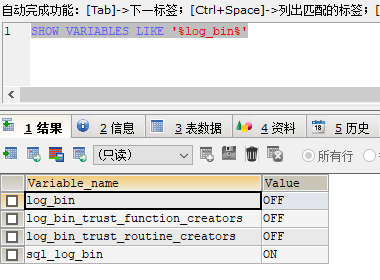
|  |
| --- |
| SELECT 'U' AS OprType, t2.id AS id, t2. NAME AS `name`  FROM snapshot\_1 tl INNER JOIN snapshot\_2 t2 ON t1. id = t2.1  WHERE tl NAME <> a t2. NAME  SELECT 'D' AS OprType, tl id AS id, t1 NAME AS `name`  FROM SNAPSHOT 1 tl LEFT JOIN snapshot\_2 t22 ON t1.id=t2.id  WHERE t2. id IS NULL  UNION ALL  SELECT 'I' AS OprType, t2.Id AS id, L2. NAME AS `name`  FROM snapshot\_2 t2 LEFT JOIN snapshot\_1 t1 ON t2.id =t  WHERE tl id IS NULL; |

就可以获取到数据，这种方式需要临时过渡区来保存快照，浪费存储空间，另外数据量很大时，这种查询会有严重的性能问题。

* 基于日志的CDC

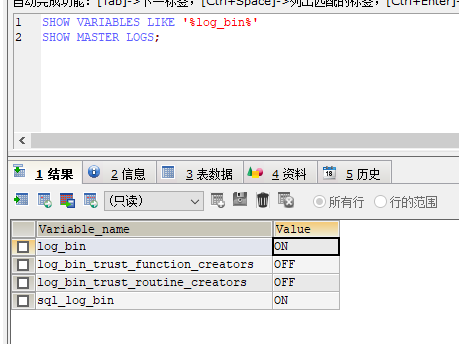
最复杂和最没有入侵的CDC方式就是基于日志的方式。数据库会把每个插入、更新和删除操作记录在日志里。只要在数据库中开启事务日志，再将日志读取出来，就可以还原这部分变化数据了。例如在mysql中，启用log\_bin日志，并使用mysqlbinlog工具读取出来

首先查看一下是否开启了二进制日志：SHOW VARIABLES LIKE '%log\_bin%'



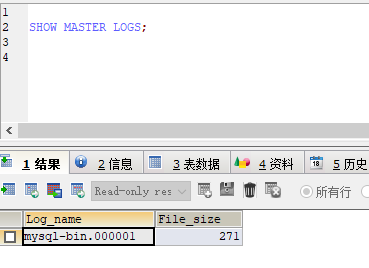
off没有开启，开启步骤如下

1 找到mysql的配置文件，my.cnf一般在/etc，在[mysqld]下面添加log-bin= /var/log/mysql-bin，重启mysql服务



on变为开启

|  |
| --- |
| 插入一条数据  INSERT INTO dim\_date VALUES (NULL,'2050-01-02',1,'January',1,2050)  查看当前日志  show master status; 或者 SHOW MASTER LOGS; |



查看日志，在shell下执行

|  |
| --- |
| mysqlbinlog --start-position=0 mysql-bin.000001 | mysql -uroot -p123456  把日志文件输出到文本文件中  mysqlbinlog –start-date=’2018-11-25’ –end-date=’2018-11-26’ mysql-bin.000001 > /tmp/000001.txt |

执行文本文件的sql语句就可以了。基于日志的CDC也有缺陷，即只能用来处理一种特定的数据库。

导出为文本方式

以文件文件的形式交换数据是一种可行的通用方法，一般需要将数据源库中的数据导出成预定的义好的分隔符，比如逗号分隔，然后用hadoop的dfs命令将文件上传到hive表对应目录中，或者使用hive的load data local inpath 语句将数据装载的目标表中。大多数据数据库都提供导出的工具或命令。比如mysql中，可以使用 select … into outfile

|  |
| --- |
| select \* into outfile /tmp/t1.txt’ fields terminated by ‘,’ from t1; |

如果不带查询条件或者导出整个库，可以使用mysqldump命令行工具，可以一次导出多个表，多个库或所有库的数据库。

|  |
| --- |
| mysqldump -uroot -p123456 –databases test > /tmp/test.sql |

使用sqoop抽取数据

Sqoop是用来实现结构型数据（如关系数据库）和Hadoop之间进行数据迁移的工具,我们可以使用sqoop来对我们的销售案例进行数据抽取

|  |  |  |  |
| --- | --- | --- | --- |
| 源数据表 | RDS库中的表(ODS) | DW库中的表 | 抽取模式 |
| customer | customer | dim\_customer | 整体拉取 |
| product | product | dim\_product | 整体拉取 |
| sales\_order | sales\_order | fact\_sales\_order dim\_order | 基于时间戳CDC拉取 |

1 覆盖导入

对于 customer、 product这两个表采用整体拉取的方式抽取数据。ETL通常是按一个固定的时间间隔,周期性定时执行,因此对于整体拉取的方式而言,每次导入的数据需要覆盖上次导入的数据。Sqoop中提供了 hive-overwrite参数实现覆盖导入。 hive-overwrite的另一个作用是提供了一个幂等操作的选择。所谓幂等操作指的是其执行任意多次所产生的影响均与次执行的影响相同。这样就能在导入失败或修复bug后可以再次执行该操作,而不用担心重复执行会对系统造成数据混乱。具体Sqoop命令如下,其中 hive-import参数表示向hive表导入,hive- table参数指定目标hive库表。

|  |
| --- |
| sqoop import --connect jdbc:mysql://192.168.25.120:3306/sales\_source --username root --password 123456 --table customer --hive-import --hive-table sales\_rds.customer --hive-overwrite --target-dir temp  sqoop import --connect jdbc:mysql://192.168.25.120:3306/sales\_source --username root --password 123456 --table product --hive-import --hive-table sales\_rds.product --hive-overwrite --target-dir temp |

注意，1) 这个脚本里面是里面的命令没有加路径，需要将sqoop的路径配到环境变量中

2) 要设置target-dir不然会报目录存在的错误

3）出错的解决方案：

Caused by: java.lang.ClassNotFoundException: org.apache.hadoop.hive.conf.HiveConf

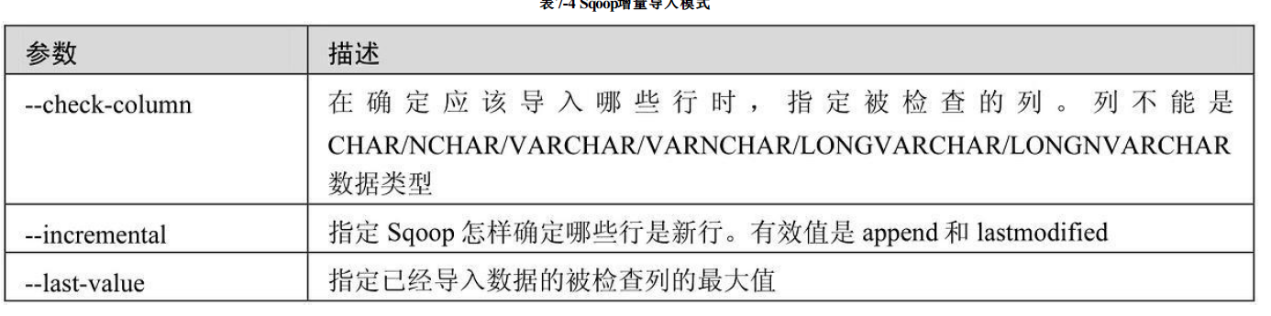
复制hive lib 包下面的hive-common-1.1.0-cdh5.13.2.jar 至sqoop lib 包下

Caused by: java.lang.ClassNotFoundException: org.apache.hadoop.hive.shims.ShimLoader

拷贝 hive lib 包下 hive-exec-1.1.0-cdh5.13.2.jar 至 sqoop 的lib包下

2 增量导入

Sqoop提供增量导入模式，用于只导入新的数据行, Sqoop支持两种类型的增量导入：append和lastmodified。可以使用--incremental参数指定增量导入的类型。



Sqoop支持两种类型的增量导入：append和lastmodified。可以使用--incremental参数指定增量导入的类型。

当被导入表的新行具有持续递增的行id值时，应该使用append模式。指定行id为--check-column的列。Sqoop导入那些被检查列的值比--last-value给出最大的数据行。

Sqoop支持的另一个表修改策略叫做lastmodified模式。当源表的数据行可能被修改，并且每次修改都会更新一个last-modified列为当前时间戳时，应该使用lastmodified模

式。那些被检查列的时间戳比--last-value给出的时间戳新的数据行被导入。

增量导入命令执行后，在控制台输出的最后部分，会打印出后续导入需要使用的last-value。当周期性执行导入时，应该用这种方式指定--last-value参数的值，以确保只导入新的或修改过的数据。可以通过一个增量导入的保存作业自动执行这个过程，这是适合重复执行增量导入的方式。

有了对Sqoop增量导入的基本了解，下面看一下如何在本示例中使用它抽取数据。对于sales\_order这个表采用基于时间戳的CDC拉取方式抽取数据。这里假设源系统中

销售订单记录一旦入库就不再改变，或者可以忽略改变，也就是说销售订单是一个随时间变化单向追加数据的表。sales\_order表中有两个关于时间的字段，order\_date表示

订单时间，entry\_date表示订单数据实际插入表里的时间，在后面11.5节讨论“迟到的事实”时就会看到两个时间可能不同。那么用哪个字段作为CDC的时间戳呢？设想这样

的情况，一个销售订单的订单时间是2018年1月1日，实际插入表里的时间是2018年1月2日，ETL每天0点执行，抽取前一天的数据。如果按order\_date抽取数据，条件为

where order\_date >= '2018-01-02' AND order\_date < '2018-01-03'，则2018年1月3日0点执行的ETL不会捕获到这个新增的订单数据。所以应该以entry\_date作为CDC的时间戳。

下面测试一下增量导入：

1. 建立sqoop增量导入作业。

|  |
| --- |
| sqoop job --create myjob -- import --connect jdbc:mysql://192.168.25.120:3306/sales\_source \  --username root \  --password 123456 \  --query "select \* from sales\_order where \$CONDITIONS" \  --hive-import \  --hive-database sales\_rds \  --hive-table sales\_order \  --split-by order\_number \  --fields-terminated-by '\t' \  --lines-terminated-by '\n' \  --target-dir /hive/warehouse/sales\_rds.db/sales\_order \  --hive-delims-replacement ' ' \  --incremental append \  --check-column entry\_date \  --password-file /sqoop/pwd/sqoopPWD.pwd  --last-value '1900-1-1' |

查看任务

|  |
| --- |
| sqoop job --list  sqoop job --show myjob |

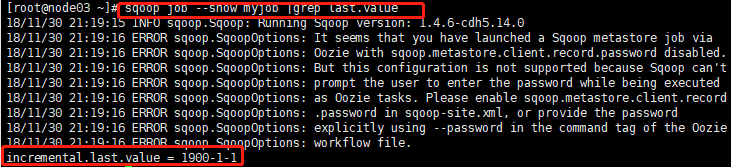
说明： 上面的语句建立一个名为myjob的Sqoop作业。使用--where参数是为了只导入前一天的数据。例如，在2点执行此作业，则不会导入0点到2点这两个小时产生的销售订单数据。

sqoop执行任务的时候会提示执行密码，需要建立密码文件并指定--password-file 文件，文件需要存放在hdfs上，权限为400

|  |
| --- |
| echo -n "123456" >sqoopPWD.pwd  hdfs dfs -mkdir -p /sqoop/pwd  hdfs dfs -put sqoopPWD.pwd /sqoop/pwd  hdfs dfs -chmod 400 /sqoop/pwd/sqoopPWD.pwd  sqoop的job中去掉--password增加   --password-file /sqoop/pwd/sqoopPWD.pwd |

|  |
| --- |
| sqoop job --create myjob -- import --connect jdbc:mysql://192.168.25.120:3306/sales\_source \  --username root \  --password-file /sqoop/pwd/sqoopPWD.pwd \  --query "select \* from sales\_order where \$CONDITIONS" \  --hive-import \  --hive-database sales\_rds \  --hive-table sales\_order \  --split-by order\_number \  --fields-terminated-by '\t' \  --lines-terminated-by '\n' \  --target-dir /hive/warehouse/sales\_rds.db/sales\_order \  --hive-delims-replacement ' ' \  --incremental append \  --check-column entry\_date \  --last-value '1900-1-1' \ |

1. 查看此时作业中保存的last-value，结果如下所示。

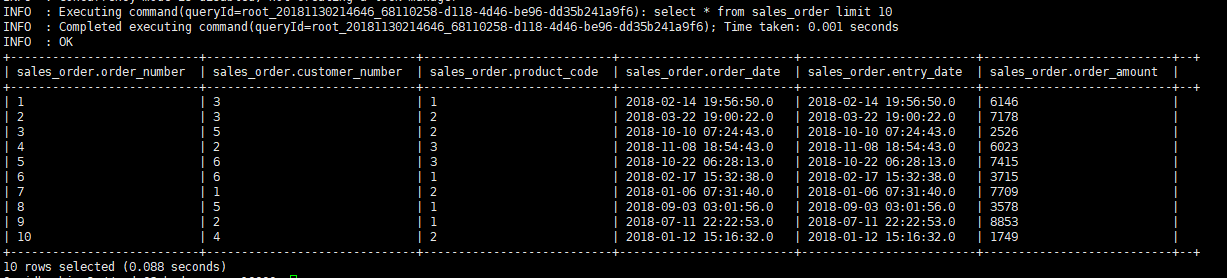


可以看到，last-value的值为初始的'1900-01-01'。

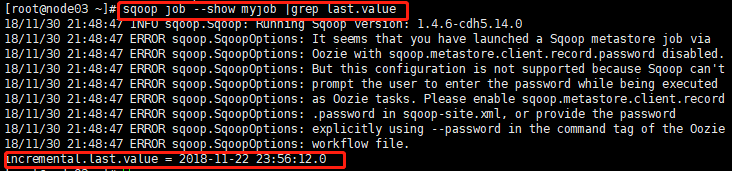
（3）首次执行作业。

|  |
| --- |
| sqoop job --exec myjob |

因为last-value的值为'1900-01-01'，所以这次会导入全部数据，查询rds.sales\_order，最后结果如下所示。



(4) 在查看此时作业中保存的last-value，结果如下所示。



(5) 可以看到，last-value的值为当前最大值'2018-11-22 23:56:12.0'。

源库增加两条数据。

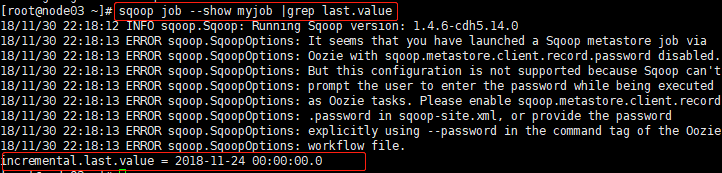
|  |
| --- |
| USE sales\_source;  SET @customer\_number := FLOOR(1 + RAND()\* 6);  SET @product\_code := FLOOR(1 + RAND()\* 2);  SET @order\_date := '2018-11-23' ;  SET @amount := FLOOR(100003 + RAND()\* 9000);  INSERT INTO sales\_order  VALUES(100003,@customer\_number,@product\_code,@order\_date,@order\_date,@amount);  SET @customer\_number := FLOOR(1 + RAND()\* 6);  SET@product\_code := FLOOR(1 + RAND()\* 2);  SET @order\_date := '2018-11-24';  SET @amount := FLOOR(1000 + RAND()\* 9000);  INSERT INTO sales\_order  VALUES(100004,@customer\_number,@product\_code,@order\_date,@order\_date,@amount);  COMMIT; |

上面的语句向sales\_order插入了两条记录，分别是11-23日的和11-24。

（6）再次执行sqoop作业，因为last-value的值为'2018-11-22 23:56:12.0'，所以这次只会导入entry\_date比'2018-11-22 23:56:12.0'大的数据。

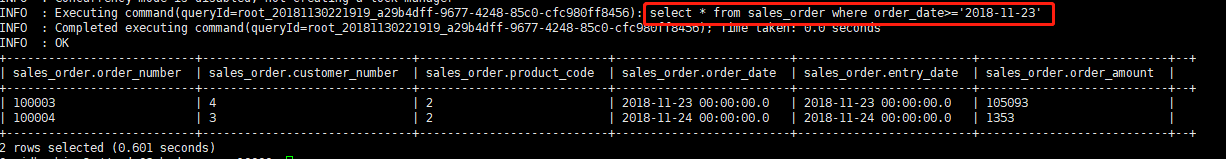
|  |
| --- |
| sqoop job --exec myjob |

（7）查看此时作业中保存的last-value，结果如下所示。



可以看到，last-value的值已经变为'2018-11-24'

（8）在hive的rds库里查询，前两行记录如下所示。



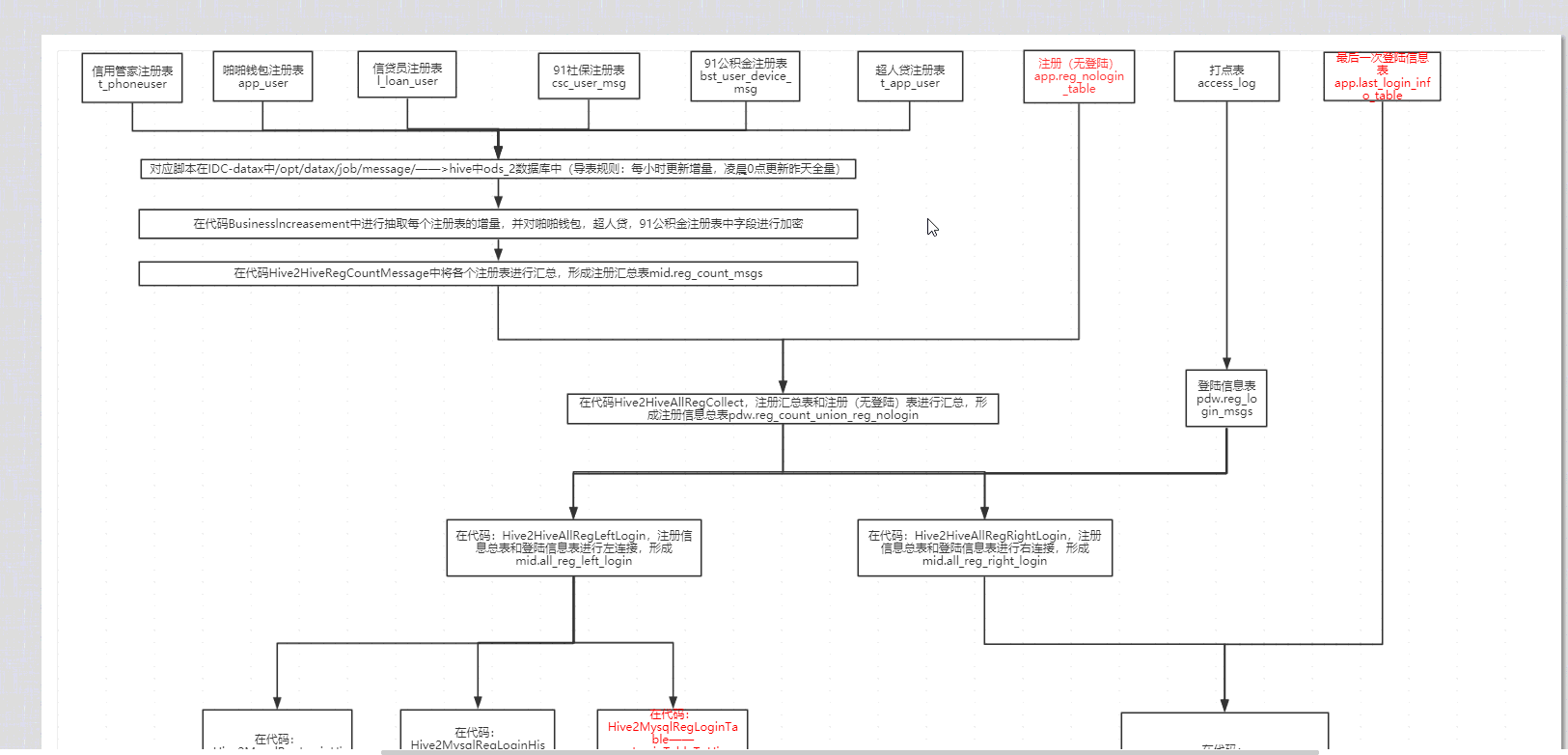
可以看到rds.sales\_order表中只新增了两条数据

现在我们已经测试了用Sqoop执行数据抽取的过程，下一节将把sqoop命令放到一个shell脚本文件中，并使用Hive进行数据的转换与装载。

###### Transform数据转换

数据转换是将数据进行重构以及标准化，消除数据的不一致，处理缺失数据，转换最主要的任务就是数据清洗。数据清洗是对数据进行重新审查和校验的过程，目的在于删除重复信息，纠正存在的错误，并提供数据一致性。我们说数据仓库的数据源是多个业务系统，各来源的数据存在着差异和和冲突，也就是我们所说的脏数据，按照一定的规则处理脏数据的过程就是数据清洗。

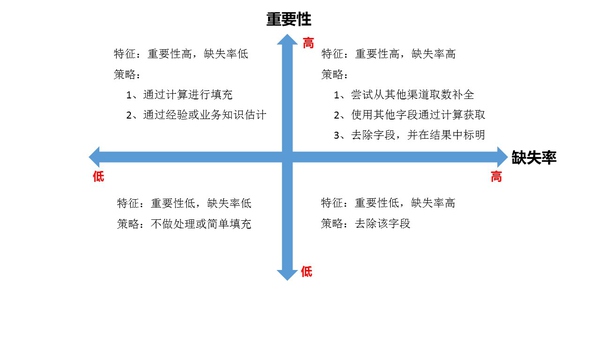
在做数据清洗的时候首先要进行清洗流程分析和设计，找到数据的主要问题和特征，画逻辑图，类似下图，将每步操作都清楚记录下来，方便项目组内的各人员的分工合作以及问题追踪



一般数据清洗流程通常包括如下内容：

* 预处理：对于大的数据文件的加载，特别是新的文件，要进行预先诊断和尖刺，不能贸然加载。
* 标准化处理：建立标准化对照表，将不一致的数据进行统一，一般需要建立标准码表，建立与标准值转化有关的函数和子程序，建立标准值与非标准值对照的映像表
* 去重处理：一种是整行数据完全重复，这种可以通过SQL语句达到去重，使用distinct或者group by 进行处理，还有一种是有重复的字段，这种一般需要子查询来进行处理，但是需要跟业务方进行确认如何保留去重数据的规则，数据清洗是一个反复的过程，需要不断的发现问题，解决问题。对于是否过滤，是否修正一般要求业务方确认，对于过滤掉的数据，写入Excel文件或者将过滤数据写入数据表，在ETL开发的初期可以每天向业务单位发送过滤数据的邮件，促使他们尽快地修正错误,同时也可以做为将来验证数据的依据。数据清洗需要注意的是不要将有用的数据过滤掉，对于每个过滤规则认真进行验证，并要业务方确认。
* 错误值：错误值产生的原因是业务系统不够健全，在接收输入后没有进行判断直接写入后台数据库造成的，比如数值数据输成全角数字字符、字符串数据后面有一个回车操作、日期格式不正确、日期越界等。这一类数据也要分类，对于类似于全角字符、数据前后有不可见字符的问题，通过转换进行处理，对于值的格式不正确或者越界或者一些主外键对应不上数据，需要将这些数据写入日志文件中，跟业务方确定这些数据的处理方式，是丢弃，还是修正，还是标准化等。
* 缺失值：缺失值的处理是很常规的操作，可以按照一下步骤进行

1、确定缺失值范围：对每个字段都计算其缺失值比例，然后按照缺失比例和字段重要性，分别制定策略，可用下图表示：



2、去除不需要的字段：这一步很简单，直接删掉即可，但强烈建议清洗每做一步都备份一下，或者在小规模数据上试验成功再处理全量数据，不然删错了会追悔莫及(多说一句，写SQL的时候delete一定要配where!)。

3、填充缺失内容：某些缺失值可以进行填充，方法有以下三种：

* 以业务知识或经验推测填充缺失值
* 以同一指标的计算结果(均值、中位数、众数等)填充缺失值
* 以不同指标的计算结果填充缺失值

前两种方法比较好理解。关于第三种方法，举个最简单的例子：年龄字段缺失，但是有身份证号，可以通过截取身份证号来获取年龄

4、重新取数：如果某些指标非常重要又缺失率高，那就需要和取数人员或业务人员了解，是否有其他渠道可以取到相关数据。

* 格式内容清洗

如果数据是由系统日志而来，那么通常在格式和内容方面，会与元数据的描述一致。而如果数据是由人工收集或用户填写而来，则有很大可能性在格式和内容上存在一些问题，简单来说，格式内容问题有以下几类：

1、时间、日期、数值、全半角等显示格式不一致

这种问题通常与输入端有关，在整合多来源数据时也有可能遇到，将其处理成一致的某种格式即可。

2、内容中有不该存在的字符

某些内容可能只包括一部分字符，比如身份证号是数字+字母，中国人姓名是汉字(赵C这种情况还是少数)。最典型的就是头、尾、中间的空格，也可能出现姓名中存在数字符号、身份证号中出现汉字等问题。这种情况下，需要以半自动校验半人工方式来找出可能存在的问题，并去除不需要的字符。

3、内容与该字段应有内容不符

姓名写了性别，身份证号写了手机号等等，均属这种问题。 但该问题特殊性在于：并不能简单的以删除来处理，因为成因有可能是人工填写错误，也有可能是前端没有校验，还有可能是导入数据时部分或全部存在列没有对齐的问题，因此要详细识别问题类型。

格式内容问题是比较细节的问题，比如同一个值，有空格和没空格统计出来结果就不正确了、统计值不全(数字里掺个字母当然求和时结果有问题)、模型输出失败或效果不好(数据对错列了，把日期和年龄混了等)。因此，在处理的数据是人工收集而来，或者产品前端校验设计不太好的时候，就需要找数据的特征，针对性的进行处理

* 逻辑错误清洗

这部分的工作是去掉一些使用简单逻辑推理就可以直接发现问题的数据，防止分析结果走偏。主要包含以下几个步骤：

去除不合理值，比如年龄超过200岁，日期越界，这种的就要么删掉，要么按缺失值处理。

* 修正矛盾内容

有些字段是可以互相验证的，比如身份证号和年龄，当年龄跟身份证号上的出生日期不匹配的时候，在这种时候，需要根据字段的数据来源，来判定哪个字段提供的信息更为可靠，去除或重构不可靠的字段。逻辑错误除了以上列举的情况，还有很多未列举的情况，在实际操作中要酌情处理。另外，这一步骤在之后的数据分析建模过程中有可能重复，因为即使问题很简单，也并非所有问题都能够一次找出，我们能做的是使用工具和方法，尽量减少问题出现的可能性，使分析过程更为高效。

* 非需求数据清洗

这一步说起来非常简单：把不要的字段删了。但实际操作起来，有很多问题，把看上去不需要但实际上对业务很重要的字段删了;某个字段觉得有用，但又没想好怎么用，不知道是否该删; 如果数据量没有大到不删字段就没办法处理的程度，那么能不删的字段尽量不删，另外必须要删的时候，一定要做好数据的备份。

* 关联性验证

如果你的数据有多个来源，那么有必要进行关联性验证。发现这种不一致，需要跟业务方确认，如何需要调整或去除数据。多个来源的数据整合是非常复杂的工作，一定要注意数据之间的关联性，尽量在分析过程中不要出现数据之间互相矛盾，而你却毫无察觉的情况，数据是数据仓库的基础，如果数据不准确，最后数据仓库中建立的数据模型也是无效的，这个数据仓库的项目就失败了，所以数据清洗特别重要，要反复验证。

###### Load 数据装载

数据仓库搭建完成后，有一些预装载的操作先进行，比如时间维度以及其他不从源数据获取的一些数据进行预装载，接着就是初始装载和定期装载。

首先我们先要将流程整理清楚

初始装载

1. 编写数据源映射关系表

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 业务表 | 源数据 | 源类型 | 贴源层 | 类型 | 数仓 | 类型 |
| 客户表 | customer | mysql表 | rds.customer | Hive表 | dw.dim\_customer | hive表 |
| 产品表 | product | mysql表 | rds. product | Hive表 | dw.dim\_product | Hive表 |
| 销售订单 | Sales\_order | mysql表 | rds.sales\_order | Hive表 | dw.dim\_order  dw.fact\_sales\_order | Hive表 |

1. 确定加载过程中需要做的处理过程，进行实现，本例中需要生成sk键

sk的生成方式有两种，使用row\_number()函数，或者创建hive的用户自定义函数，关联一个函数row\_sequence()，分别演示演示一下这两种方式，然后确定一种方式。

* + 假如有一个表t和一个维度表dim\_t，使用row\_number（）的方式生成sk

select row\_number() over(order by tid)+max\_sk,\*

from t

cross join (select coalesce(max(sk),0) from dim\_t) d

* + 使用UDF

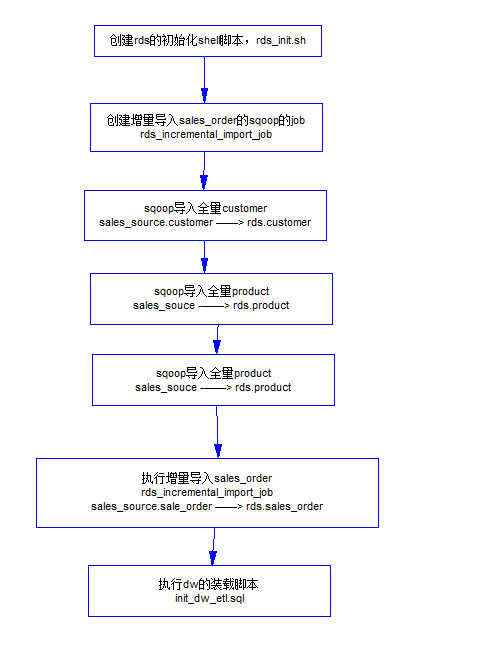
|  |
| --- |
| package org.rowsequence;    import org.apache.hadoop.hive.ql.exec.Description;  import org.apache.hadoop.hive.ql.exec.UDF;  import org.apache.hadoop.hive.ql.udf.UDFType;  import org.apache.hadoop.io.LongWritable;    /\*\*  \* UDFRowSequence.  \*/  @Description(name = "row\_sequence",  value = "\_FUNC\_() - Returns a generated row sequence number starting from 1")  @UDFType(deterministic = false)  public class RowSequence extends UDF {  private LongWritable result = new LongWritable();    public RowSequence() {  result.set(0);  }    public LongWritable evaluate() {  result.set(result.get() + 1);  return result;  }  add jar /home/hadoop/hive\_study/hive\_udf/hive\_udf.jar  create temporary function row\_sequence as 'org.rowsequence.RowSequence';  select row\_sequence(),\* from t |

* + 在hive中添加jar包，将它上传到hdfs上，添加jar包添加hdfs上的

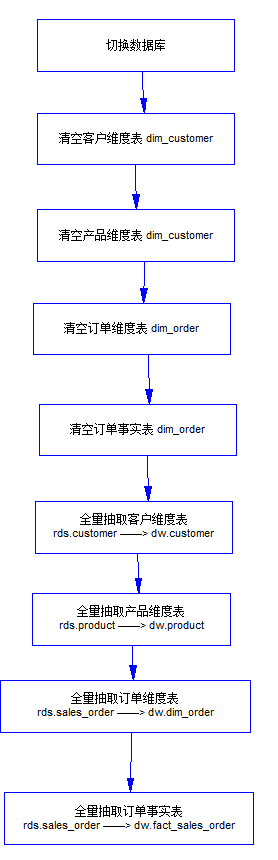
|  |
| --- |
| add jar ${hive-home}/lib/hive-contrib-1.1.0-cdh5.14.0.jar;  create function row\_sequence as 'org.apache.hadoop.hive.contrib.udf.UDFRowSequence'; |

这两种方式都可以实现，我们使用UDF的这种方式，创建一个函数row\_sequence，后面的语句中直接调用，减少sql的复杂度

1. 装载流程



dw装载子流程，init\_dw\_etl.sql



1. 这个过程需要更新hive中的表，因此要设置hive支持事务，进行行级更新，编辑hive-site.xml

<property>

<name>hive.optimize.sort.dynamic.partition</name>

<value>false</value>

</property>

<property>

<name>hive.support.concurrency</name>

<value>true</value>

</property>

<property>

<name>hive.enforce.bucketing</name>

<value>true</value>

</property>

<property>

<name>hive.exec.dynamic.partition.mode</name>

<value>nonstrict</value>

</property>

<property>

<name>hive.txn.manager</name>

<value>org.apache.hadoop.hive.ql.lockmgr.DbTxnManager</value>

</property>

<property>

<name>hive.compactor.initiator.on</name>

<value>true</value>

</property>

<property>

<name>hive.compactor.worker.threads</name>

<value>1</value>

</property>

<property>

<name>hive.in.test</name>

<value>true</value>

</property>

1. 脚本编写

* 初始化dw库的脚本init\_dw\_etl.sql

|  |
| --- |
| USE dw;  -- 清空表  TRUNCATE TABLE dim\_customer;  TRUNCATE TABLE dim\_product;  TRUNCATE TABLE dim\_order;  TRUNCATE TABLE fact\_sales\_order;  -- 装载客户维度表  INSERT INTO dim\_customer (customer\_sk,customer\_number,customer\_name,customer\_street\_address,customer\_zip\_code,customer\_city,customer\_state,`version`,effective\_date,expiry\_date)  SELECT  t1.customer\_number,  t1.customer\_number,  t1.customer\_name,  t1.customer\_street\_address,  t1.customer\_zip\_code,  t1.customer\_city,  t1.customer\_state,  1,  '2018-01-01',  '2050-01-01'  FROM sales\_rds.customer t1;    -- 装载产品维度表  INSERT INTO dim\_product (product\_sk,product\_code,product\_name,product\_category,`version`,effective\_date,expiry\_date)  SELECT t1.product\_code,  product\_code,  product\_name,  product\_category,  1,  '2016-03-01',  '2050-01-01'  FROM sales\_rds.product t1;    -- 装载订单维度表  INSERT INTO dim\_order(order\_sk,order\_number,`version`,effective\_date,expiry\_date)  SELECT t1.order\_number,  order\_number,  1,  order\_date,  '2050-01-01'  FROM sales\_rds.sales\_order t1;    -- 装载销售订单事实表  INSERT INTO fact\_sales\_order  SELECT order\_sk, customer\_sk, product\_sk, date\_sk, order\_amount  FROM sales\_rds.sales\_order a  JOIN dim\_order b ON a.order\_number = b.order\_number  JOIN dim\_customer c ON a.customer\_number = c.customer\_number  JOIN dim\_product d ON a.product\_code = d.product\_code  JOIN dim\_date e ON to\_date(a.order\_date) = e.`date`; |

* 装载数据总的shell脚本init\_all\_etl.sh

|  |
| --- |
| #!/bin/bash  # 建立Sqoop增量导入作业，以entry\_date作为检查列，初始的last-value是1900-1-1  sqoop job --create myjob -- import --connect jdbc:mysql://192.168.25.120:3306/sales\_source \  --username root \  --password-file /sqoop/pwd/sqoopPWD.pwd \  --query "select \* from sales\_order where \$CONDITIONS" \  --hive-import \  --hive-database sales\_rds \  --hive-table sales\_order \  --split-by order\_number \  --fields-terminated-by '\t' \  --lines-terminated-by '\n' \  --target-dir /hive/warehouse/sales\_rds.db/sales\_order \  --hive-delims-replacement ' ' \  --incremental append \  --check-column entry\_date \  --last-value '1900-1-1' \  # 首次抽取，将全部数据导入RDS库  sqoop import --connect jdbc:mysql://192.168.25.120:3306/sales\_source --username root --password 123456 --table customer --hive-import --hive-table sales\_rds.customer --hive-overwrite --target-dir temp  sqoop import --connect jdbc:mysql://192.168.25.120:3306/sales\_source --username root --password 123456 --table product --hive-import --hive-table sales\_rds.product --hive-overwrite --target-dir temp  # 清空rds层的订单表  beeline -u jdbc:hive2://node03.hadoop.com:10000/dw -e "TRUNCATE TABLE sales\_rds.sales\_order"  # 执行rds层的增量抽取  sqoop job --exec myjob  # 装载dw层  beeline -u jdbc:hive2://node03.hadoop.com:10000/dw -f /work/sql/init\_dw\_etl.sql |

1. 检验数据

装载数据以后要检查一下数据的正确性，源数据的条数，抽取的条数,拿customer举例

|  |
| --- |
| select count(1) from sales\_source.customer  select count(1) from rds.customer  select count(1) from dw.dim\_customer |

如果条数一致可以选几条数据查看是否都存在，一般条数一致基本上就问题不大，如果条数不一致，就需要查看差异在什么地方，查找出错的原因，调整脚本。

定期装载

1. 定期装载设计表

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 源 | Rds | 数仓 | 抽取模式 | 维度处理 |
| customer | customer | dim\_customer | 全量拉取 | address上SCD2，name上SCD1 |
| product | product | dim\_product | 全量拉取 | 均SCD2 |
| sales\_order | sales\_order | dim\_order | 每天CDC | 唯一订单号 |
| Fact\_sales\_order | 每天CDC | N/A |
| N/A | N/A | dim\_date | N/A | 预装载 |

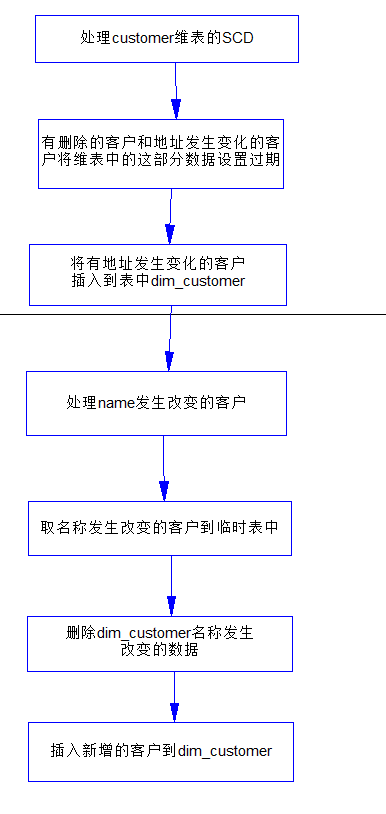
1. 确认细节实现

本例中设计到细节处理的一是缓慢变化维SCD的处理方式，二是事实表的CDC方式，customer的name属性采用SCD1的方式，即覆盖，customer\_street\_address采用SCD2方式，也就是保存历史变化。dim\_order和fact\_sales\_order采用基于时间戳的CDC装载模式，也就是使用插入时间和更新时间来捕获数据的变化，很多情况下也可以使用自增id识别插入数据。因此要记录本次增量的时间范围，我们需要设置一个表来记录上次同步数据的最大时间作为本次的开始时间，我们同步的是前一天的时间，我们的粒度是天，因此结束时间是当前日期-1天，这个表我们叫做cdc\_time,有两个字段，start\_date date,end\_date date。

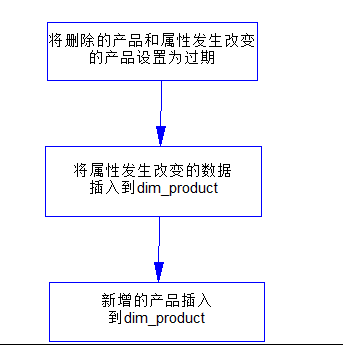
|  |
| --- |
| create table sales\_rds.cdc\_time(start\_date date,end\_date date);  insert into cdc\_time values("2018-11-30","2018-12-1"); |

1. 流程图

处理dim\_customer的SCD



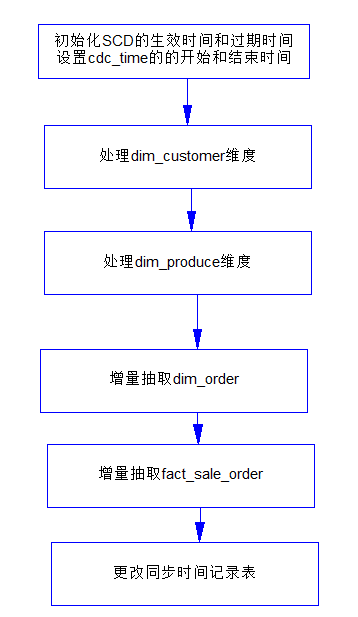
处理dim\_product



处理订单维度表dim\_order,直接抽取增量数据插入即可

处理订单事实表fact\_sales\_order,直接抽取即可。

整体流程图如下



1. 脚本编写 schedule\_daily\_etl.sql

|  |
| --- |
| -- 设置scd的生效时间和过期时间  SET hivevar:cur\_date = CURRENT\_DATE();  SET hivevar:pre\_date = DATE\_ADD(${hivevar:cur\_date},-1);  SET hivevar:max\_date = CAST('2050-01-01' AS DATE);  -- 设置cdc的开始结束日期  INSERT overwrite TABLE sales\_rds.cdc\_time  SELECT end\_date, ${hivevar:cur\_date} FROM sales\_rds.cdc\_time;  -- 装载customer维度  -- 获取源数据中被删除的客户和地址发生改变的客户，将这些数据设置为过期时间，即当前时间的前一天  UPDATE dim\_customer  SET expiry\_date = ${hivevar:pre\_date}  WHERE dim\_customer.customer\_sk IN(SELECT  a.customer\_sk  FROM (SELECT  customer\_sk,  customer\_number,  customer\_street\_address  FROM dim\_customer  WHERE expiry\_date = ${hivevar:max\_date}) a  LEFT JOIN sales\_rds.customer b ON a.customer\_number = b.customer\_number  WHERE b.customer\_number IS NULL  OR a.customer\_street\_address <> b.customer\_street\_address);  -- 将有地址变化的插入到dim\_customer表，如果有相同数据存在有不过期的数据则不插入  INSERT INTO dim\_customer  SELECT t1.customer\_number,  t1.customer\_number,  t1.customer\_name,  t1.customer\_street\_address,  t1.customer\_zip\_code,  t1.customer\_city,  t1.customer\_state,  t1.version,  t1.effective\_date,  t1.expiry\_date  FROM(SELECT  t2.customer\_number customer\_number,  t2.customer\_name customer\_name,  t2.customer\_street\_address customer\_street\_address,  t2.customer\_zip\_code,  t2.customer\_city,  t2.customer\_state,  t1.version + 1 `version`,  ${hivevar:pre\_date} effective\_date,  ${hivevar:max\_date} expiry\_date  FROM dim\_customer t1  INNER JOIN sales\_rds.customer t2 ON t1.customer\_number = t2.customer\_number  AND t1.expiry\_date = ${hivevar:pre\_date}  LEFT JOIN dim\_customer t3 ON t1.customer\_number = t3.customer\_number  AND t3.expiry\_date = ${hivevar:max\_date}  WHERE t1.customer\_street\_address <> t2.customer\_street\_address  AND t3.customer\_sk IS NULL  ) t1;    -- 处理customer\_name列上的scd1，覆盖  -- 不进行更新，将源数据中的name列有变化的数据提取出来，放入临时表  -- 将 dim\_couster中这些数据删除、  -- 将临时表中的数据插入  DROP TABLE IF EXISTS tmp;  CREATE TABLE tmp AS  SELECT a.customer\_sk,  a.customer\_number,  b.customer\_name,  a.customer\_street\_address,  a.customer\_zip\_code,  a.customer\_city,  a.customer\_state,  a.version,  a.effective\_date,  a.expiry\_date  FROM dim\_customer a  JOIN sales\_rds.customer b ON a.customer\_number = b.customer\_number  Where a.customer\_name <> b.customer\_name ;  -- 删除数据  DELETE FROM  dim\_customer WHERE  dim\_customer.customer\_sk IN (SELECT customer\_sk FROM tmp);  -- 插入数据  INSERT INTO dim\_customer  SELECT \* FROM tmp;  -- 处理新增的customer记录  INSERT INTO dim\_customer  SELECT t1.customer\_number,  t1.customer\_number,  t1.customer\_name,  t1.customer\_street\_address,  t1.customer\_zip\_code,  t1.customer\_city,  t1.customer\_state,  1,  ${hivevar:pre\_date},  ${hivevar:max\_date}  FROM( SELECT t1.\*  FROM sales\_rds.customer t1  LEFT JOIN dim\_customer t2 ON t1.customer\_number = t2.customer\_number  WHERE t2.customer\_sk IS NULL ) t1;  -- 装载product维度  -- 取源数据中删除或者属性发生变化的产品，将对应  UPDATE dim\_product  SET expiry\_date = ${hivevar:pre\_date}  WHERE dim\_product.product\_sk IN(SELECT a.product\_sk  FROM(SELECT product\_sk,  product\_code,  product\_name,  product\_category  FROM dim\_product  WHERE expiry\_date = ${hivevar:max\_date}) a  LEFT JOIN sales\_rds.product b ON a.product\_code = b.product\_code  WHERE b.product\_code IS NULL  OR (a.product\_name <> b.product\_name OR a.product\_category <> b.product\_category));    -- 处理product\_name、product\_category列上scd2的新增行  INSERT INTO dim\_product  SELECT ORDER BY t1.product\_code,  t1.product\_code,  t1.product\_name,  t1.product\_category,  t1.version,  t1.effective\_date,  t1.expiry\_date  FROM( SELECT t2.product\_code product\_code,  t2.product\_name product\_name,  t2.product\_category product\_category,  t1.version + 1 `version`,  ${hivevar:pre\_date} effective\_date,  ${hivevar:max\_date} expiry\_date  FROM dim\_product t1  INNER JOIN sales\_rds.product t2 ON t1.product\_code = t2.product\_code  AND t1.expiry\_date = ${hivevar:pre\_date}  LEFT JOIN dim\_product t3 ON t1.product\_code = t3.product\_code  AND t3.expiry\_date = ${hivevar:max\_date}  WHERE(t1.product\_name <> t2.product\_name  OR t1.product\_category <> t2.product\_category)  AND t3.product\_sk IS NULL  ) t1;    -- 处理新增的 product 记录  INSERT INTO dim\_product  SELECT t1.product\_code,  t1.product\_code,  t1.product\_name,  t1.product\_category,  1,  ${hivevar:pre\_date},  ${hivevar:max\_date}  FROM( SELECT t1.\*  FROM sales\_rds.product t1  LEFT JOIN dim\_product t2 ON t1.product\_code = t2.product\_code  WHERE t2.product\_sk IS NULL  ) t1;  -- 装载order维度  INSERT INTO dim\_order  SELECT t1.order\_number,  t1.order\_number,  t1.version,  t1.effective\_date,  t1.expiry\_date  FROM( SELECT order\_number order\_number,  1 `version`,  order\_date effective\_date,  '2050-01-01' expiry\_date  FROM sales\_rds.sales\_order, rds.cdc\_time  WHERE entry\_date >= last\_load AND entry\_date < current\_load ) t1;  -- 装载销售订单事实表  INSERT INTO sales\_fact\_sales\_order  SELECT order\_sk,  customer\_sk,  product\_sk,  date\_sk,  order\_amount  FROM sales\_rds.sales\_order a,  dim\_order b,  dim\_customer c,  dim\_product d,  date\_dim e,  rds.cdc\_time f  WHERE a.order\_number = b.order\_number  AND a.customer\_number = c.customer\_number  AND a.order\_date >= c.effective\_date  AND a.order\_date < c.expiry\_date  AND a.product\_code = d.product\_code  AND a.order\_date >= d.effective\_date  AND a.order\_date < d.expiry\_date  AND to\_date(a.order\_date) = e.date  AND a.entry\_date >= f.start\_date  AND a.entry\_date < f.end\_date ; |

shell脚本 schedule\_daily.sh

|  |
| --- |
| #!/bin/bash  # 整体拉取rds层customer、product表数据  sqoop import --connect jdbc:mysql://192.168.25.120:3306/sales\_source --username root --password 123456 --table customer --hive-import --hive-table sales\_rds.customer --hive-overwrite --target-dir temp  sqoop import --connect jdbc:mysql://192.168.25.120:3306/sales\_source --username root --password 123456 --table product --hive-import --hive-table sales\_rds.product --hive-overwrite --target-dir temp  # 执行增量导入  sqoop job --exec myjob  # 调用 schedule\_daily\_etl.sql 文件执行定期装载  beeline -u jdbc:hive2://node03.hadoop.com:10000/dw -f /work/sql/schedule\_daily\_etl.sql |

1. 检验数据

当任务开始跑以后,要验证几天的数据，看数据是否正确，如果数据存在问题，可能是脚本存在bug，需要做调整

###### ETL自动化

上面我们已经把每天需要增量跑的任务编写完了，这个任务需要每天定时去执行，一般来讲需要制定一个任务计划表，这个任务来记录所有的任务的执行周期，执行时间，负责人，联系方式等信息，当一个项目很大，有很多的任务，每个任务的调度周期和时间可能是不一样的，比如有日结的数据，也有月结的数据，日结是每天要进行汇总计算的，月结是每月月底要进行计算的，可能日结是项目组中的一个人写的，月结是令一个人写的，为保证整体项目的在试运行期间的平稳运行，一般要有这么一个表格，当项目有问题时，找相应的负责人。

我们的示例比较简单，只有一个调度任务

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 功能模块 | 调度文件 | 调度周期 | 负责人 | 联系方式 |
| 增量抽取数据 | schedule\_daily.sh | 每天凌晨2点 | 张三 | 13912345678 |

关于调度的执行周期，一般选在业务量的低峰，上线也是一样，在业务量最低的时候去进行操作，这个时候如果出现问题，进行回滚，业务影响不大，所以一般选择在凌晨2-4点之间，关于上线也多说一句，一般来讲公司有几个环境，开发环境，测试环境和生产环境。一般我们在开发的时候使用的数据库连的都是开发环境，开发环境一般性能不高，数据库里面的数据也是样本数据，量很小，只是为了开发使用，测试环境一般是为了模拟上线用的，预上线就是上线到测试环境，测试环境一般来讲跟生产环境是相同的，性能跟生产环境不能说一模一样，至少也是要差不多，开发人员开发完成后需要预上线到测试环境，测试人员就开始对系统进行功能测试和压力测试，并且编写测试报告，测试报告通过，才能正式上线，上线需要跟运维提邮件申请，各负责人进行审批，运维人员进行上线，上线以后，测试人员需要进行拨测，一般来说上线的时候测试和开发以及运维都需要在现场，一旦发生问题，马上回滚。上线通常在周二和周四的晚上12点以后，避开业务的高峰期，一般选择凌晨2点左右，如果发生回滚，在4点以前搞不定，这次上线就失败了，底下去修复bug，等下次上线周期。回滚机制是必须要有的，如果上线失败导致业务出现大量的问题，这个时候就发生了生产事故，事故一般会定级，当级别比较高的时候，会进行邮件的通报，甚至全公司通报，每个审批的负责人都要进行相应的处理比如扣绩效罚款等，开发人员也会相应的处罚但是没有负责人承担的重，所以其实一旦发生问题程序员就要跑路真实况发生的几率并不大，除非是毁灭性的不可恢复的，否则发生问题各级的负责人会受到处罚，还落不到程序员头上，但是负责人被处罚了，部门可能会秋后算账，你的年终绩效可能要受到影响，这就看老大的处事风格了，大家了解一下即可。

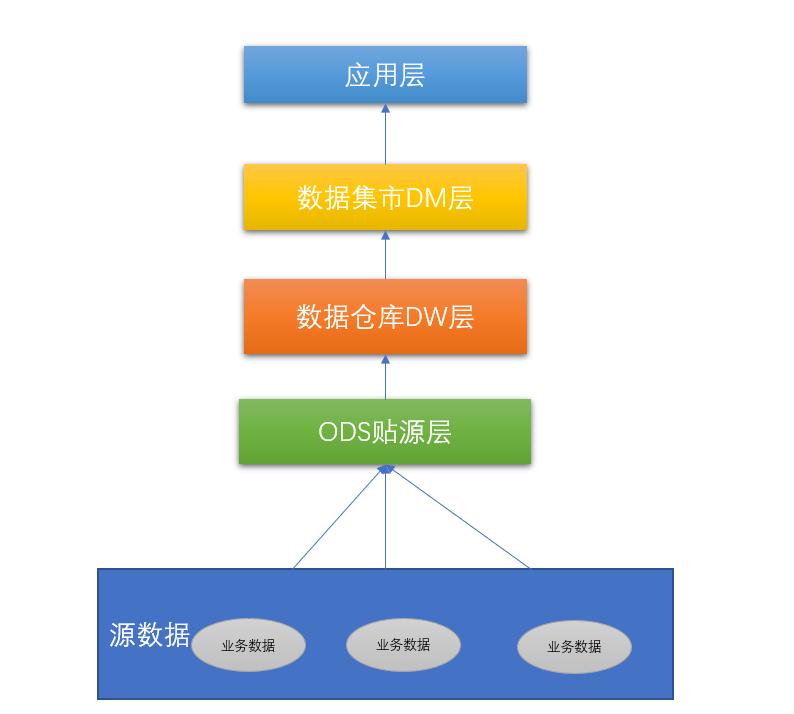
定时执行任务最简单的就是使用crontab，我们使用crontab命令来创建一个任务，定期运行脚本。

|  |
| --- |
| # 修改文件属性为可执行  chmod 755 root/ schedule\_daily.sh  # 编辑crontab文件内容  crontab -e  # 打开编辑页面，插入一行，指定每天2点执行定期装载作业，然后保存退出  0 2 \* \* \* /root/ schedule\_daily.sh |

也可以使用oozie这样的框架进行可视化的这种工作流方式进行任务调度。需要说明的是如果oozie来调度sqoop，需要开启sqoop的元数据共享。

###### 数据集市DM层

数据集市：是按照主题域组织的数据集合，用于支持部门级的决策。数据集市的类型有两种：独立数据集市和从属数据集市。独立数据集市集中于部门所关心的单一主题域，数据以部门为基础部署，不考虑企业级别的信息共享。从属数据集市，数据来源于数据仓库，数据仓库里的数据经过整合、重构、汇总后传递给数据集市。我们一般用聚合数据放入到数据集市中。我们的入门案例，日统计数据和月统计数据和年统计数据放入dm层，因此我们的数据仓库的数据从源数据层（source）🡪贴元层（ods、rds）🡪数仓层（dw）🡪数据集市层（dm）🡪mysql🡪报表展示。



##### 数据可视化

报表的展示有很多方式echart，或者java的进行开发都可以，我们也可以使用zeppelin来进行报表的展示，对于展示来讲虽然是数仓的一部分，通常是由前端的开发人员来进行开发的，我们只需要将需要展示的数据组织好，跟前端人员协调好数据接口，前端取数据进行展示就可以了，所以这部分大家了解即可。

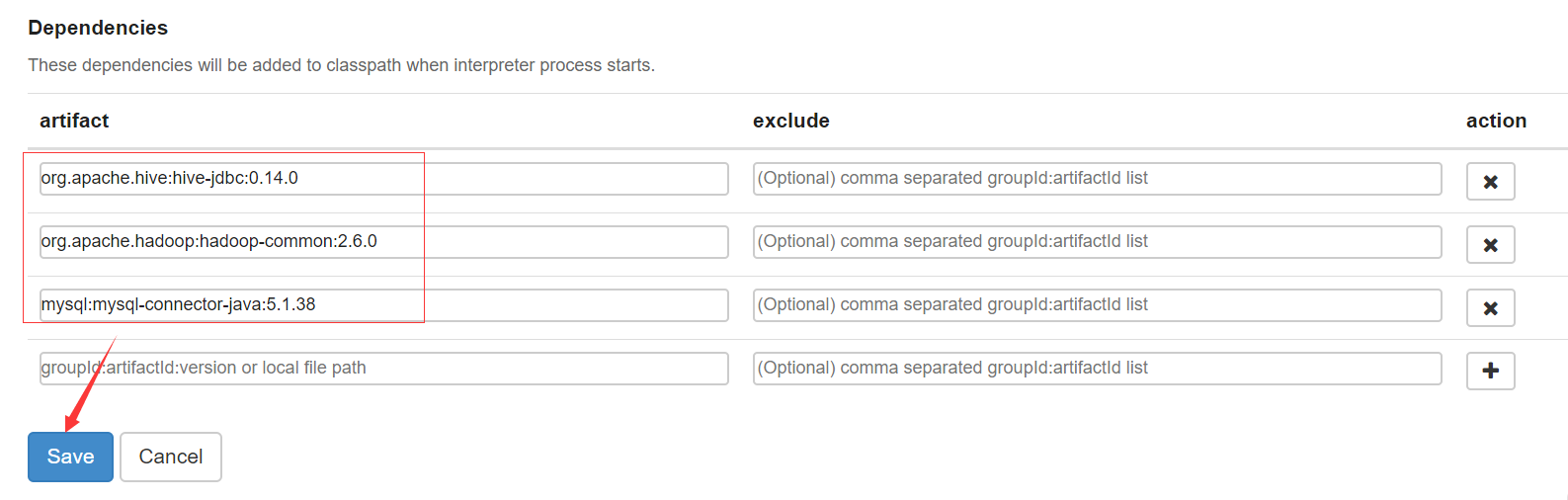
Apache Zeppelin [1] 是一个让交互式数据分析变得可行的基于网页的开源框架。Zeppelin提供了数据分析、数据可视化等功能。

Zeppelin 是一个提供交互数据分析且基于Web的笔记本。方便你做出可数据驱动的、可交互且可协作的精美文档，并且支持多种语言，包括 Scala(使用 Apache Spark)、Python(Apache Spark)、SparkSQL、 Hive、 Markdown、Shell等等。

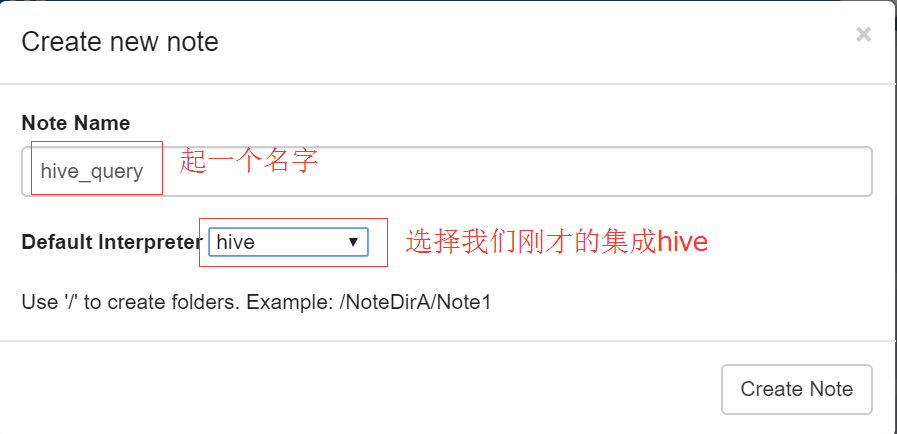
配置连接hive



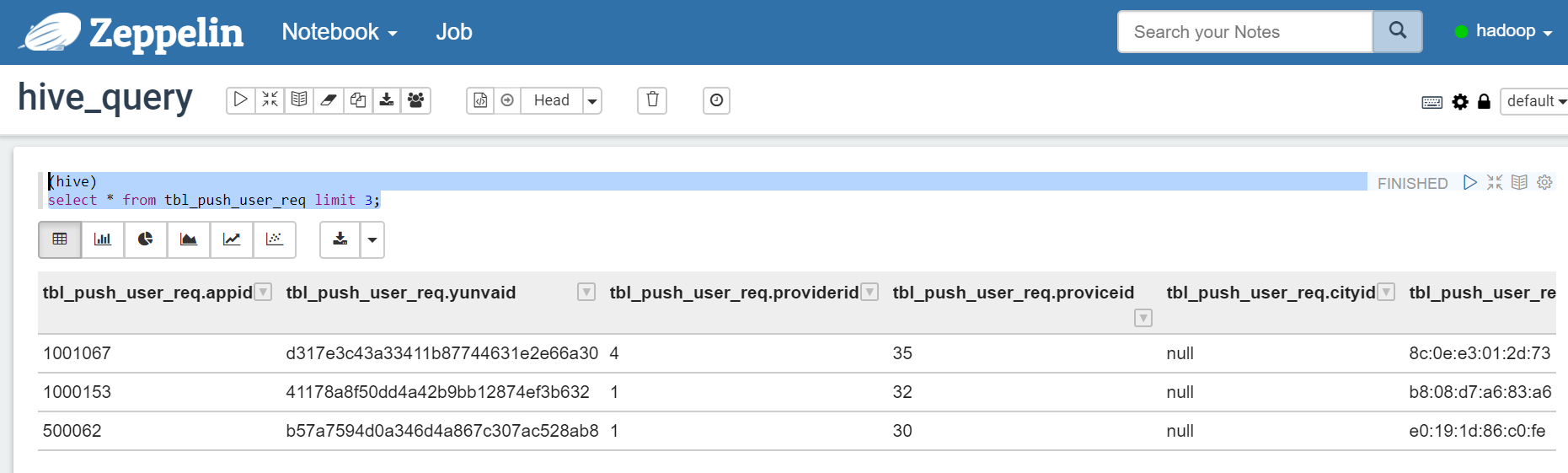




创建一个note



执行一个查询语句



显示图表

