**Mycat跨分片Join指南**

# 前言

Mycat目前版本支持跨分片的join,主要实现的方式有四种。

全局表

ER分片

HBT(参考MyCAT人工智能解决跨分片SQL.docx)

ShareJoin

ShareJoin在开发版中支持，前面三种方式1.3.0.1支持

# ShareJoin

ShareJoin是一个简单的跨分片Join,基于HBT的方式实现。

目前支持2个表的join,原理就是解析SQL语句，拆分成单表的SQL语句执行，然后把各个节点的数据汇集。

支持任意配置的A,B表

如：

A,B的dataNode相同

<table name="A" dataNode="dn1,dn2,dn3" rule="auto-sharding-long" />

<table name="B" dataNode="dn1,dn2,dn3" rule="auto-sharding-long" />

A,B的dataNode不同

<table name="A" dataNode="dn1,dn2 " rule="auto-sharding-long" />

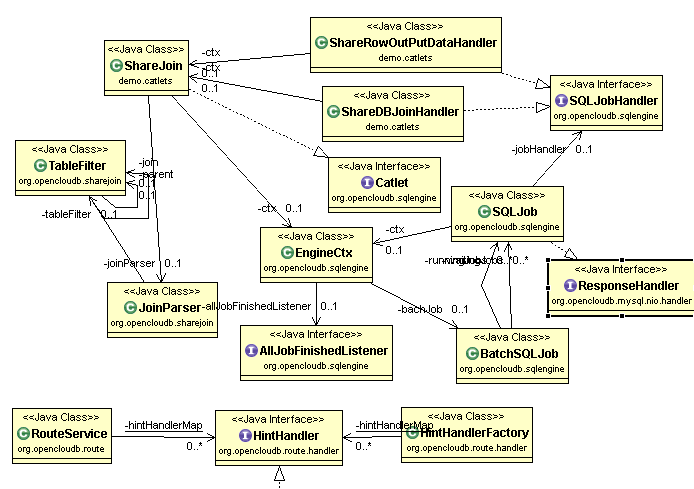
<table name="B" dataNode="dn1,dn2,dn3" rule="auto-sharding-long" />

或

<table name="A" dataNode="dn1 " rule="auto-sharding-long" />

<table name="B" dataNode=" dn2,dn3" rule="auto-sharding-long" />

## 相关类图



JoinParser: SQL语句的解析

TableFilter:存解析后的各个子表

ShareJoin:执行拆分的语句管理控制,和字段，记录的管理

ShareDBJoinHandler:第一个表执行后获取数据的handler

ShareRowOutPutDataHandler：最后一个表执行后获取数据的handler

EnginerCtx：执行引擎

SQLJob：SQL语句执行任务

SQLJobHandler：SQL语句执行后获取数据的handler

BatchSQLJob：批量执行任务控制

AllJobFinishedListener：所有任务完成侦听器

HintCatletHandler

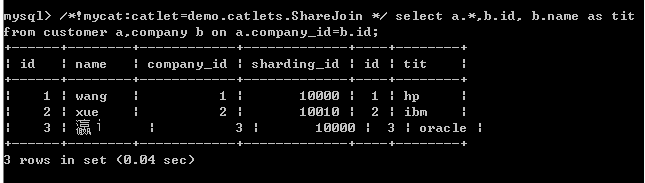
ShareJoin

RouteService

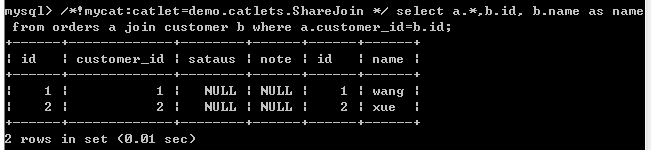
## 测试

默认mycat的环境测试：

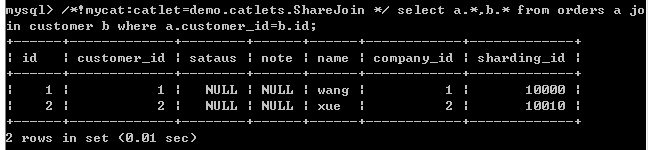
/\*!mycat:catlet=demo.catlets.ShareJoin \*/ select a.\*,b.id, b.name as tit from customer a,company b on a.company\_id=b.id;



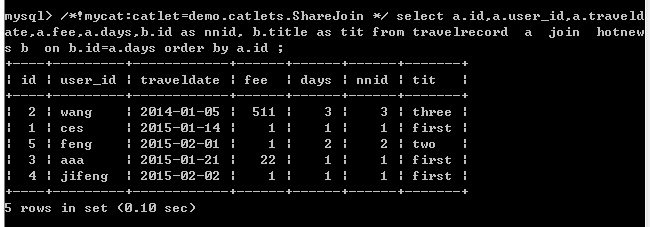
/\*!mycat:catlet=demo.catlets.ShareJoin \*/ select a.\*,b.id, b.name as name from orders a join customer b where a.customer\_id=b.id;



/\*!mycat:catlet=demo.catlets.ShareJoin \*/ select a.\*,b.\* from orders a join customer b where a.customer\_id=b.id;



/\*!mycat:catlet=demo.catlets.ShareJoin \*/ select a.id,a.user\_id,a.traveldate,a.fee,a.days,b.id as nnid, b.title as tit from travelrecord a join hotnews b on b.id=a.days order by a.id ;



## 升级

未来支持多表的跨分片Join

小表放人缓存或广播方式

# 全局表

一个真实的业务系统中，往往存在大量的类似字典表的表格，它们与业务表之间可能有关系，这种关系，可以理解为“标签”，而不应理解为通常的“主从关系”，这些表基本上很少变动，可以根据主键ID进行缓存，下面这张图说明了一个典型的“标签关系”图：

**网络设备**

在分片的情况下，当业务表因为规模而进行分片以后，业务表与这些附属的字典表之间的关联，就成了比较棘手的问题，考虑到字典表具有以下几个特性：

* 变动不频繁
* 数据量总体变化不大
* 数据规模不大，很少有超过数十万条记录。

鉴于此，**MyCAT**定义了一种特殊的表，称之为“全局表”，全局表具有以下特性：

* 全局表的插入、更新操作会实时在所有节点上执行，保持各个分片的数据一致性
* 全局表的查询操作，只从一个节点获取
* 全局表可以跟任何一个表进行JOIN操作

将字典表或者符合字典表特性的一些表定义为全局表，则从另外一个方面，很好的解决了数据JOIN的难题。通过全局表+基于E-R关系的分片策略，**MyCAT**可以满足80%以上的企业应用开发。

## 配置

全局表配置比较简单，不用写Rule规则，如下配置即可：

<table name="company" primaryKey="ID" type="global" dataNode="dn1,dn2,dn3" />

需要注意的是，全局表每个分片节点上都要有运行创建表的DDL语句。

# ER分片

**MyCAT**借鉴了NewSQL领域的新秀Foundation DB的设计思路，Foundation DB创新性的提出了Table Group的概念，其将子表的存储位置依赖于主表，并且物理上紧邻存放，因此彻底解决了JION的效率和性能问题，根据这一思路，提出了基于E-R关系的数据分片策略，子表的记录与所关联的父表记录存放在同一个数据分片上。

**customer**采用sharding-by-intfile这个分片策略，分片在dn1,dn2上，**orders**依赖父表进行分片，两个表的关联关系为orders.customer\_id=customer.id。于是数据分片和存储的示意图如下：

Dn2(orders)

Parent\_Id:3

Parent\_id :4

Dn1(orders)

Parent\_id:1

Parent\_id:2

Dn2(customer)

Id:3

Id:4

Dn1(customer)

Id:1

Id:2

orders

customer

这样一来，分片Dn1上的的customer与Dn1上的orders就可以进行局部的JOIN联合，Dn2上也如此，再合并两个节点的数据即可完成整体的JOIN，试想一下，每个分片上orders表有100万条，则10个分片就有1个亿，基于E-R映射的数据分片模式，基本上解决了80%以上的企业应用所面临的问题。

## 配置

以上述例子为例，schema.xml中定义如下的分片配置：

<table name="**customer**" dataNode="dn1,dn2" rule="sharding-by-intfile">

<childTable name="**orders**" joinKey="customer\_id" parentKey="id"/>

</table>

# HBT分片

解决跨分片的SQL JOIN的问题，远比想象的复杂，而且往往无法实现高效的处理，既然如此，就依靠人工的智力，去编程解决业务系统中特定几个必须跨分片的SQL的JOIN逻辑，MyCAT提供特定的API供程序员调用，这就是MyCAT创新性的思路——人工智能。

以一个跨节点的SQL为例，

Select a.id,a.name,b.title from a,b where a.id=b.id

其中a在分片1，2，3上，b在4，5，6上，需要把数据全部拉到本地（MyCAT服务器）,执行JOIN逻辑，具体过程如下（只是一种可能的执行逻辑）：

EngineCtx ctx=new EngineCtx();//包含MyCat.SQLEngine

String sql=,“select a.id ,a.name from a ”;

//在a表所在的所有分片上顺序执行下面的本地SQL

ctx.executeNativeSQLSequnceJob(allAnodes,new DirectDBJoinHandler());

DirectDBJoinHandler类是一个回调类，负责处理SQL执行过程中返回的数据包，这里的这个类，主要目的是用a表返回的ID信息，去b表上查询对于的记录，做实时的关联：

DirectDBJoinHandler{

Private HashMap<byte[],byte[]> rows;//Key为id,value为一行记录的Column原始Byte数组，这里是a.id,a.name,b.title这三个要输出的字段

Public Boolean onHeader(byte[] header)

{

//保存Header信息，用于从Row中获取Field字段值

}

Public Boolean onRowData(byte[] rowData)

{

String id=getColumnAsString(“id”);

//放入结果集,b.title字段未知，所以先空着

rows.put(getColumnRawBytes(“id”),rowData);

//满1000条，发送一个查询请求

String sql=”select b.id, b.name from b where id in (………….)”;

//此SQL在B的所有节点上并发执行，返回的结果直接输出到客户端

ctx.executeNativeSQLParallJob(allBNodes,sql ,new MyRowOutPutDataHandler(rows));

}

Public Boolean onRowFinished()

{

}

Public void onJobFinished()

{

If(ctx.allJobFinished())

{///used total time ….

}

}

}

/最后，增加一个Job事件监听器，这里是所有Job完成后，往客户端发送RowEnd包，结束整个流程。

ctx.setJobEventListener(new JobEventHandler(){public void onJobFinished(){ client.writeRowEndPackage()}});

以上提供一个SQL执行框架，完全是异步的模式执行，并且以后会提供更多高质量的API，简化分布式数据处理，比如内存结合文件的数据JOIN算法，分组算法，排序算法等等，

期待更多的牛人一起来完善。

## 配置

无