中国矿业大学计算机学院

2020 级本科生课程实验报告

课程名称：	大数据存储与管理课程设计
报告时间：	2022 年 12 月 25 日
学生姓名：	朱少行
学　　号：	12204123
专业班级：	大数据 20-3 班
任课教师：	闫秋艳

3 大数据存储案例设计

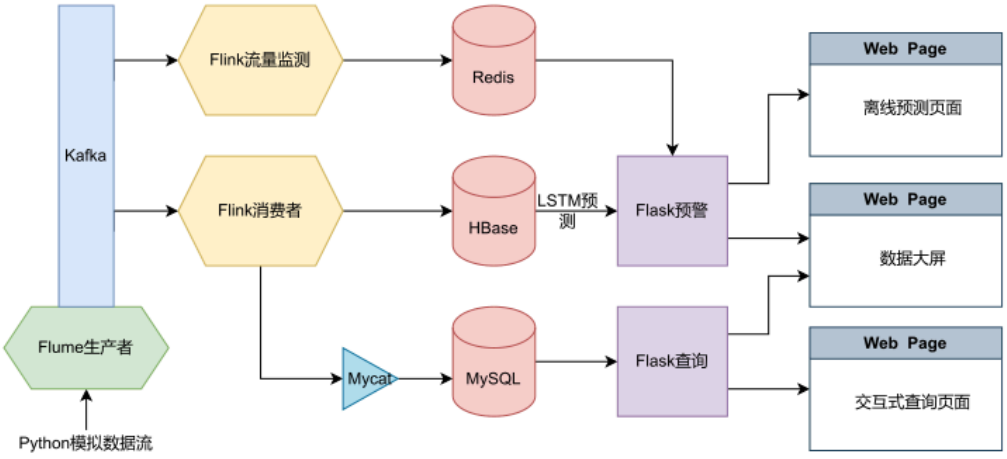
3.1 实验目标

通过课程学习所得分布式数据库知识基础，结合数据可视化技术及大数据架构知识，搭建一个分布式的数据的交通可视化平台。

3.2 实验内容

- 数据大屏。从不同维度统计数据，并通过多种图表展示。
- 交互式查询。自定义条件查询数据，并将结果展示或导出。
- 流量报警。监测一段时间内的车流量，并进行实时报警。
- 流量预测。根据历史车流量数据，预测未来车流量。

3.3 系统整体架构



3.4 系统实现（个人部分）

运行环境：Ubuntu20.04

运行平台：华为云

3.4.1 MySQL 安装

安装 mysql5.7，实现远程登录（node3 为主节点）

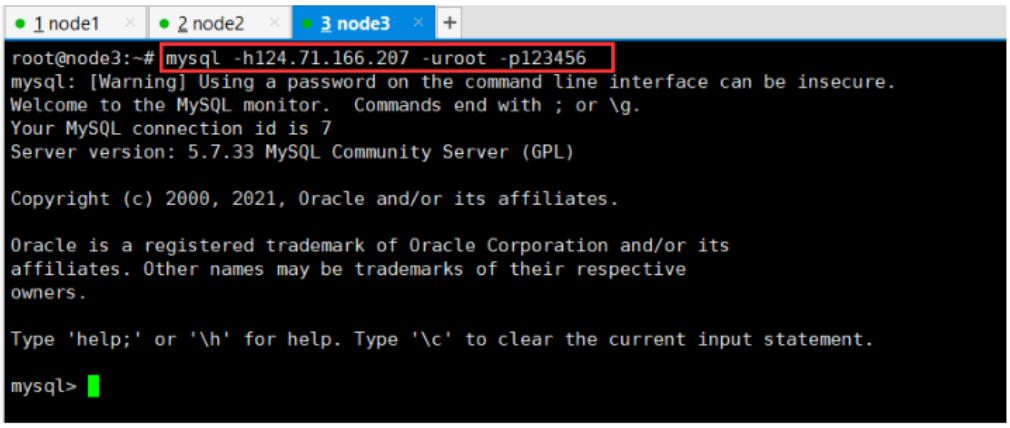


图 1 远程登录 node1

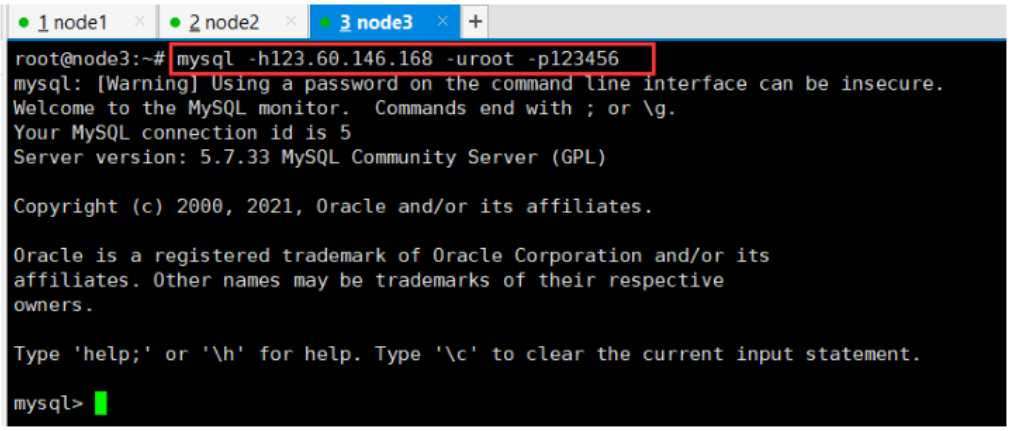
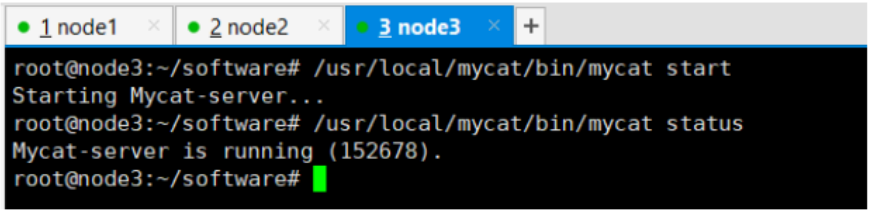


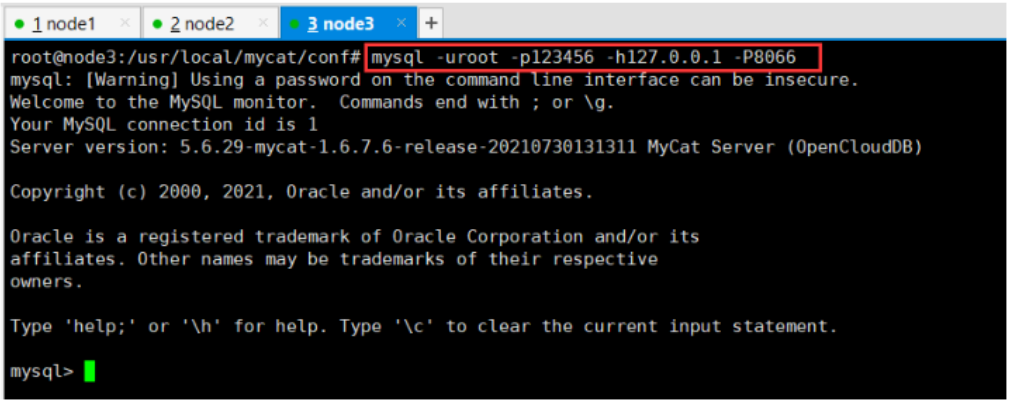
图 2 远程登录 node2

3.4.2 MyCat 集群搭建

步骤一 安装 MyCat



步骤二 登录 MyCat 服务器



步骤三 MyCat 读写分离、主从复制搭建

MyCat 集群搭建情况

- ✧ node3 主节点，node1 为从节点
- ✧ node3，node2 数据为数据节点
- ✧ node3 与 node1 实现主从复制和读写分离

3.4.3 分库分表规则

首先列举出数据字段信息如下：

序号	字段名称	字段描述
1	XH	序号
2	CX	车型
3	SFZCKMC	收费站出口名称
4	CKSJ	出口时间
5	SFZRKMC	收费站入口名称
6	RKSJ	入口时间
7	BZ	备注
8	CP	车牌

数据分析

在 178396 条数据中，主要分为两大部分——深圳出和深圳入。深圳入有 86637 条，深圳出有 91759 条。

收费站出口 SFZCKMC 字段取值情况有四种，当数据为深圳入时取值为广东罗田主线站、广东水朗 D 站、松山湖南，在 86637 条深圳入数据中，广东罗田主线站为 62948 条，广东水朗 D 站为 18495 条，松山湖南为 5194 条。剩余的 91759 条深圳出数据中均为 NULL。

收费站入口 SFZRKMC 字段共有 1233 种不同取值（除去字段值为未知的情况）

序号 XH 字段不存在重复情况，可以用该字段作为主键。

车牌号 CP 字段，存在重复情况。

在深圳出的数据中收费站出口 SFZCKMC 字段和出口时间 CKSJ 字段均为 NULL，同时在深圳出的数据中，车牌 CP 字段也存在未知的情况。

BZ	CP	计数项:BZ
深圳出		463
	\N	463
总计		463

BZ	计数项:BZ
深圳出	91759
深圳入	86637
总计	178396

SFZCKMC	计数项:SFZRKMC
广东罗田主线站	62948
广东水朗D站	18495
松山湖南	5194
未知	91759
总计	178396

业务逻辑分析

在高速公路 ETC 大数据管理平台中，通常情况下，我们要做的就是统计高速公路整体车流量、统计某个站点的车流量、高速公路上车辆的型号、车辆在深圳市的出入情况。

我们通常不会查询某一辆车的出入记录，除非在某些特殊的情况下。这决定了车牌信息在高速公路 ETC 大数据管理平台中意义不大。

解决方案-1

针对车辆在深圳市的出入情况，首先将表做水平分片，分为深圳出、深圳入两部分。

针对高速公路车辆的型号，除了型号字段，我们也不需要其他字段。

针对整体车流量的统计，我们需要的仅仅是数据表中条目的数量，因此不需

要额外的字段信息。因此我们可以将上面两个需求结合起来，设置表*etc_inout*，字段为“XH、CX、CP、BZ”，其中 XH 为主键和外键，首先进行垂直分片、然后根据深圳出入信息进行水平分片。

针对某个站点的车流量，需要对应站点的信息。所以需要对站点进行水平划分，因为收费站入口站点非常多，有 1233 个站点，而收费站出口站点比较少，只有三个。如果按照收费站出口站点进行水平划分，那么每当查询入口站点的时候，都需要查询所有结点上的数据，因为入口站点比较多，那么就会产生大量的传输代价。如果按照收费站入口站点进行水平划分，因为出口站点只有三个，即使全部查询，也只需要 3 次完全传输代价，相较于 1233 次完全传输代价，3 次传输代价已经很少了。因此选择入口站点作为水平划分依据，设置表*etc_station*，字段为“SFZRKMC、SFZCKMC”

因为进行增量查询时需要时间字段，所以每一个表中都需要含有时间字段 CKSJ、RKSJ。

所以最后表的划分情况如表 1 所示，两个垂直分片直接使用序号字段 XH 进行连接，但是在实际应用中不需要进行连接，因为每一个垂直分片都可以独立完成相应业务逻辑。

表 1 混合分片示意图-1

<table>
<tr><th>XH</th><th>CX</th><th>CP</th><th>BZ</th><th>RKSJ</th><th>CKSJ</th><th>SFZRKMC</th><th>RKSJ</th><th>SFZCKMC</th><th>CKSJ</th></tr>
<tr><td colspan="6" rowspan="3">深圳入

深圳出</td><td colspan="4">入口站点分片 1</td></tr>
<tr><td colspan="4">入口站点分片 2</td></tr>
<tr><td colspan="4">…</td></tr>
</table>

分库规则

对于*etc*_inout表，采用枚举分片*sharding-by-intfile*，深圳入的数据存放在第一个节点上，深圳出的数据存放在第二个节点上

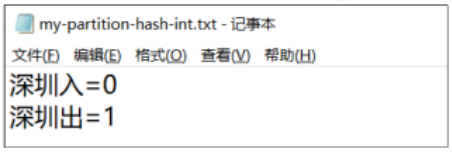


图 3 my-partition-hash-int.txt 文件

```
<tableRule name="my-sharding-by-intfile">
    <rule>
        <columns>BZ</columns>
        <algorithm>my-hash-int</algorithm>
    </rule>
</tableRule>

<function>
    <function name="my-hash-int"
          class="io.mycat.route.function.PartitionByFileMap">
    <property name="mapFile">my-partition-hash-int.txt</property>
    <property name="type">1</property> <!-- 1表示字段的值是字符串，0表示是数值 -->
    <property name="defaultNode">0</property><!-- 默认存放在结点0上 -->
</function>
```

图 4 定义 my-sharding-by-intfile 分片规则

对于*etc_station*表，采用一致性哈希分片*sharding-by-murmur*

```
<tableRule name="my-sharding-by-murmur">
    <rule>
        <columns>SFZRKMC</columns>
        <algorithm>murmur</algorithm>
    </rule>
</tableRule>
```

图 5 定义*my-sharding-by-murmur*分片规则

```
<schema name="MYETCDB" checkSQLschema="false" sqlMaxLimit="100" dataNode="dn1">
    <table name="etc_inout" primaryKey="XH" dataNode="dn1,dn2" rule="my-sharding-by-intfile"></table>
    <table name="etc_station" primaryKey="XH" dataNode="dn1,dn2" rule="my-sharding-by-murmur"></table>
    <table name="cp_map" primaryKey="id" dataNode="dn1,dn2" type="global"></table>
</schema>
```

图 6 *schema.xml*文件

解决方案-2

垂直分片，就是将一个表的多个字段拆分到多个表中或者是多个库上去。每个库表的数据表结构都不同，每个库表只包含部分字段。一般来说，会将较少的访问频率比较高的表放到同一张表中，然后将较多的访问频率比较低的字段放到另外一张表中。因为数据库是有缓存的，访问频率高的字段少，在缓存中存放的行越多，查询性能就越好。

但是在解决方案-1 中，所有垂直分片都是独立完成业务逻辑，分片的每一个字段都需要使用，即每一个字段的访问频率都是一样的。因为垂直分片的本意就是将访问频率较高的字段放到同一张表中、将访问频率比较低的字段放到另外一张表中以提高查询性能，所以垂直分片的意义在本实验中并没有完全体现出来。

因此解决方案-2 将表按照 SFZRKMC 字端进行水平分片，采用一致性哈希分片*sharding-by-murmur*，最后表的划分情况表 2 所示。

表 2 混合分片示意图-2

XH	CX	CP	SFZRKMC	RKSJ	SFZCKMC	CKSJ	BX
入口站点分片 1							
入口站点分片 2							
…							

分库规则

```
<tableRule name="my-sharding-by-murmur">
    <rule>
        <columns>SFZRKMC</columns>
        <algorithm>murmur</algorithm>
    </rule>
</tableRule>
```

图 7 定义*my-sharding-by-murmur*分片规则

```
<schema name="MYETCDB" checkSQLschema="false" sqlMaxLimit="100" dataNode="dn1">
    <table name="etc" primaryKey="XH" dataNode="dn1,dn2" rule="my-sharding-by-murmur"></table>
    <table name="cp_map" primaryKey="id" dataNode="dn1,dn2" type="global"></table>
</schema>
```

图 8 *schema.xml*文件

其中*schema.xml*文件中额外定义了一个表，这个是车牌与地区的映射表，将其定义为全局表，主要作用是将车牌号到对应地区，完成车辆来源地的统计。

```
mysql> select * from cp_map;
+----------+--------------------------+
| ID       | NAME                     |
+----------+--------------------------+
| 云       | 云南省                   |
| 京       | 北京市                   |
| 冀       | 河北省                   |
```

图 9 *cp_map*表部分数据

3.4.4 可视化图表

可视化图表主要分为两部分——数据统计分析图表和预警图表。

数据统计分析图表由$MyCat$提供相关数据。

3.4.5 增量查询

为了提高查询效率，全部接口都使用增量查询。

增量查询整体思想——每次仅查询上次请求到当前请求这段时间内的数据变化情况，然后维护一个全局信息表，将该段时间内的数据变化情况同步到全局信息表当中，同时将数据返回个前端。

以查询入站口车流量为例，假设两次查询时间间隔为t,全局变量表中定义当前记录时间T，该时间车流量N，当经过t时间需要更新车流量的时候，查询数据得到新增车数量为n，那么更新车流量$N = \frac{T \times N + n}{T + t}$，同时更新$T = T + t$。

核心代码主要为$solution$类和$MyDB$类，$solution$类实现增量查询，$MyDB$类实现SQL查询。

特别地，为了提高代码的复用性，将各种字段的增量查询整合为一个函数，并且使用tag进行区分。对于车牌号CP字段，因为第一个字符标志车辆来源地，所以采用$left(CP, 1)$取出该字段值的第一个字符进行统计分析。

核心代码如下：

增量查询 solution 类

```
def count(tablename, tag, factor_str = ''):
    # 正式环境下使用
    starttime = Config.time2dict[tag]
    endtime = datetime.now()
    endtime = endtime - timedelta(seconds = 1)
    if endtime < starttime:
        starttime = endtime
    Config.time2dict[tag] = endtime + timedelta(seconds = 1)

    # 测试环境下使用
    # starttime = Config.time2dict[tag]
    # endtime = starttime + timedelta(seconds = Config.interval_second - 1)
    # nowtime = datetime.now()
    # if endtime > nowtime:
    #     endtime = nowtime
    # if endtime < starttime:
    #     starttime = endtime
    # Config.time2dict[tag] = endtime + timedelta(seconds = 1)

    # 时间转字符串
    starttime_str = "'" + time2str(starttime) + "'"
    endtime_str = "'" + time2str(endtime) + "'"

    db = MyDB()
    if tag == "CP":
        res = db.select_count_part_of_key(tablename, tag, starttime_str, endtime_st
r, factor_str)
    elif tag == "D_bottom":
        res = db.select_count_part_of_key(tablename, "CX", starttime_str, endtime_s
tr, factor_str)
    else:
        res = db.select_count(tablename, tag, starttime_str, endtime_str, factor_st
r)

    tag_dict = Config.name2dict[tag]

    if tag == "D_bottom":
        type2num = Config.type2num
        h = starttime.hour
        for k, v in res:
            car_type = type2num[k]
            tag_dict[car_type][h][0] = car_type
            tag_dict[car_type][h][1] = h
```

```
            tag_dict[car_type][h][2] += v

    else:
        for k, v in res:
            Config.total2dict[tag] += v
            if k not in tag_dict:
                tag_dict[k] = v
            else:
                tag_dict[k] += v

    print("total:", Config.total2dict[tag])
    return tag_dict
```

SQL 查询 MyDB 类

```
def select_count(self, table_name, field, starttime, endtime, factor_str = ''):
    """
    增量查询各字段数量
    :param table_name: 表名
    :param field: 所要查询的字段
    :param starttime: 开始时间
    :param endtime: 结束时间
    :param factor_str: 查询条件
    :return:
    """
    if factor_str == '':
        sql = f"select {field},count({field}) from {table_name} where RKSJ between
{starttime} and {endtime} group by {field} "
    else:
        sql = f"select {field},count({field}) from {table_name} where {factor_str}
and RKSJ between {starttime} and {endtime} group by {field}"
    try:
        print(sql)
        self.cursor.execute(sql)
        res = self.cursor.fetchall()
        return res
    except Exception as e:
        print("查询失败\n", e)
```

```
def select_count_part_of_key(self, table_name, field, starttime, endtime, factor_st
r = ''):
    """ 增量查询部分字段"""
    if factor_str == '':
```

```
        sql = f"select left({field},1),count({field}) from {table_name} where RKSJ
between {starttime} and {endtime} group by left({field},1) "
    else:
        sql = f"select left({field},1),count({field}) from {table_name} where {fact
or_str} and RKSJ between {starttime} and {endtime} group by left({field},1)"
    try:
        print(sql)
        self.cursor.execute(sql)
        res = self.cursor.fetchall()
        return res
    except Exception as e:
        print("查询失败\n", e)
```

3.4.6 交互式查询

交互式查询，在查询页面设置不同查询条件供用户进行选择。查询界面主要由多个下拉框组成，下拉框中包含不同的查询条件，用户选择查询条件之后，前端将查询条件返回给后端，后端进行相应查询，并返回数据给前端。

同时可以将查询结果导出为 Excel 或 csv 文件。

核心代码主要为$utils$类，$solution$类和$MyDB$类，$utils$类为工具类，实现查询字典到查询条件的转换，$solution$类实现交互式查询，$MyDB$类实现SQL查询

特别地，为了提高代码的复用性，各类交互式查询整合为一个函数，$cond$字典为查询条件字典，$filed$为查询字段。为了提高代码的鲁棒性，对$cond$查询条件字典和$filed$查询字段进行了特殊处理，$cond$查询条件字典可以为$None$，即查询当前数据库中全部数据，$filed$查询字段默认为“$*$”，即查询所有字段$select\ *$

核心代码如下：

交互式查询 utils 类

```
def get_factor_str(self, condition: dict):
    """获得查询条件"""
    factor_str = ""
    if condition is None:
        return factor_str
    for k, v in condition.items():
        if k == "CKSJ" or k == "RKSJ":
            factor_str += k + " between " + "'" + v[0] + "'" + " and " + "'" + v[1]
 + "'" + " and "
        else:
```

```
            factor_str += k + " = " + "'" + v + "'" + " and "
    factor_str = factor_str[:-5]
    return factor_str
```

交互式查询 solution 类

```
def select_interactive(tablename, condition = None, field = "*"):
    db = MyDB()
    factor_str = db.get_factor_str(condition)
    res = db.select_interactive(tablename, factor_str, field = field)
    return res
```

SQL 查询 MyDB 类

```
def select_interactive(self, table_name, factor_str = "", field = "*"):
    """
    交互式查询
    :param table_name: 表名
    :param factor_str: 查询条件
    :param field: 查询字段,字符串形式,中间使用逗号隔开
    :return:
    """
    if factor_str == "":
        sql = f"select {field} from {table_name}"
    else:
        sql = f"select {field} from {table_name} where {factor_str}"
    try:
        print(sql)
        self.cursor.execute(sql)
        res = self.cursor.fetchall()
        return res
    except Exception as e:
        print("查询失败\n", e)
```

3.5 项目总结和反思

*MyCat*数据查询

在进行数据生成的时候，为了更好的模拟真实情况，数据采用实时生成的方法，并通过 Kafka2SQL 的方式插入数据库，因此后端在处理查询请求的时候，也是使用实时时间进行查询，但是在实际测试中发现，由于数据插入和数据查询是异步请求，同时使用增量查询，只查询某一时刻的数据，所以可能出现在某一

时刻数据查询先于数据插入执行的情况，在这种情况下查询结果为 NULL，进而导致后续一连串查询错误——后续增量查询大部分查询结果为 NULL。

出现这种情况，推测是$MyCat$并发控制出现了问题，最后解决办法选择了延时查询，其整体思想比较简单，每次收到查询请求之后，查询当前时刻前 1 秒的数据，进而保证在任一时刻数据插入先于数据查询，但是也有可能因为网络状况导致插入时间超过一秒钟，为了解决这一问题，将数据按时刻查询改为按照时间段查询，查询的时间段为从上一查询时刻到当前时刻的这段时间。

安装$MySQL$

搭建$MyCat$集群，首先需要安装$MySQL$，因为是第一次使用$Ubuntu$系统，在这里也踩了不少坑，将其总结如下：

✧ 建议不要使用 MySQL8.0

MySQL 主要分为 5.7 和 8.0 两个主要版本，但通过$Ubuntu$自带的apt 安装 MySQL 时，会自动安装最新版本，即 8.0 版本，虽然高版本有高版本的好处，但是也有其不便之处。

以配置大小写不敏感为例，根据官方文档，MySQL 8.0 以上版本大小写不敏感只能在初始化服务器时配置。禁止在服务器初始化后更改 l$ower_case_table_names$ 设置。$lower\_case\_table\_names$ 配置必须在安装好 MySQL 后，初始化 $mysql$ 配置时才有效。一旦 $mysql$启动后，再设置是无效的，而且启动报错。

因此需要重新初始化 MySQL 数据库，并且在初始化之前将 $lower_case_table_names = 1$写入到$my.cnf$文件中。重新初始化数据库就意味着备份数据库，卸载现有环境并重新安装。这个过程非常麻烦，并且 MySQL8.0 的语法与 MySQL5.7 的语法上面也有不同。

鉴于此，最终选择删除 MySQL8.0 转而安装 MySQL5.7，但是安装 MySQL 时同时安装了对应版本的依赖文件，如何彻底删除 MySQL8.0 也是一个问题。具体删除方式见附录。

✧ 安装 MySQL 5.7

因为 Ubuntu20.04 的 apt 默认选择按照高版本 MySQL，所以按照 MySQL5.7 就比较麻烦，尝试了多种方法进行安装，但是均出现各种各样的问题，比较简单的是使用 deb 安装包方式安装。具体安装方式见附录。

✧ 系统命令

在 Ubuntu 系统中，$mysql$的启动和启动不再是$systemctl\ start\ mysqld$，而是$ystemctl\ start\ mysql.service$。

同时，配置文件不再是 my.cnf（注意在$mysql$目录下依然有$my.cnf$文件），而是$mysql.conf.d$下的$mysqld.cnf$

关于$MyCat$

虽然说$MyCat$是一个中间件，比较简单，它的功能就是帮我们整合分布式$MySQL$数据库，但是在最开始进行实验的时候，也只是单纯按照实验指导书上面的步骤进行操作，对于$MyCat$逻辑数据库、逻辑表的创建并没有很深刻的理解，以至于在最开始想要进行$MyCat$的分库分表的时候，完全不知道从哪里下手。经过进一步的学习和实践，我对$MyCat$的使用也有了更加深刻的理解，现将我对$MyCat$的理解总结如下：

✧ **定义逻辑、物理数据库**

逻辑数据库通过 schema.xml 进行定义

- 逻辑数据库的名字为 MYETCDB，在<scheme name="MYETCDB">中定义
- 物理数据库的名字为 myetc，在<dataNode database="myetc">中定义

✧ **创建物理数据库**

物理数据库创建的时候，我们需要在每一个结点的$mysql$中进行创建，创建的名称与 schema.xml 中定义的一致。

✧ **表的定义**

所有表的创建全部在$mycat$中进行，在$schema.xml$中定义的表为虚拟的表，在表中只定义了表的名称，主键，存放的结点。

如果存放的结点多于 2 个，则属于水平分片，需要给出分片规则。

✧ **表的创建**

表的创建分为全局表、水平分片、内联表（导出式水平分片）

全局表和水平分片的表，需要在$mycat$中创建，$mycat$就会根据$schema.xml$中的设定的结点位置，自动在对应结点的$mysql$中进行创建。

内联表不需要创建，由$mycat$自动生成，它的使用逻辑就是当我们在$mycat$中插入数据的时候，$mycat$会自动将数据按照规则存放到对应结点的内联表中。

✧ **主从复制、读写分离**

$node1$和$node3$实现类主从复制和读写分离。

主从复制指的是$node1$为主节点、$node3$为从节点。

读写分离指的是$node1$复制写操作、$node3$复制读操作，因为二者是主从关系，所以$node3$上的数据与$node1$上的数据完全一致，但是与$node2$没有任何关系。相当于$node3$是$node1$的副本，在进行读操作的时候，$mycat$从$node2$、$node3$进行读取，进行写操作的时候，$mycat$从$node1$、$node2$进行写入，然后$node3$从$node1$进行复制。

枚举分片*sharding-by-intfile*

在最开始设置枚举分片之后，出现了插入失败的情况，报错信息为$columnValue: in\ Please\ check\ if\ the\ format\ satisfied$。当时知道是格式错误，但是排查表$etc_inout$的结构，没有任何问题，也没有数据长度超限的情况。猜想可能是枚举分片的问题，或许是$sharding\text{-}by\text{-}intfile$只支持$int$类型。

查阅资料后发现，$sharding\text{-}by\text{-}intfile$可以支持字符串类型，但是需要进行显示设置，同时为了防止非枚举值的数据出现，最好设置默认存放结点。默认存放结点的意思是说当出现不在枚举范围内的值时，将其存放在默认结点上。

```
<property name="type">1</property> <!-- 1表示字段的值是字符串，0表示是数值 -->
<property name="defaultNode">0</property><!-- 默认存放在结点0上 -->
```

3.6 结论与讨论

总的来说，在本次大数据存储案例设计中，主要使用了$flask$框架和$MyCat$分布式数据库，负责整个后端和数据库的工作，包括$MyCat$环境搭建、分库分表设计、后端API设计、增量查询、交互式查询等。

在这个过程中，一路磕磕绊绊，也踩了不少坑，遇到了很多细节性的小问题，最后也在不断的学习和查资料的过程中一一解决，虽然过程比较艰辛，但是收获也很多。事件的处理逻辑、数据查询代价、代码的简洁和复用性、应用耦合度等非常多的问题都需要考虑和解决，对我来说也是一次很好的历练机会。同时，在这个过程中我也对实际生产环境下的数据库系统操作有了更深的理解和认识。

最后，再来说一下我对《大数据存储与管理实验》这门课的感受。这门课是实践性质的课，结合课堂上的理论学习，使得我对$MyCat$数据库和$Hbase$数据库有了更进一步的认识，而不是仅仅停留在理论层面。让我印象最深刻的是最后的大数据存储案例设计，在小组成员的合作下，我们真正的完成了一个可以称得上

是项目的项目。也是在这个过程中，小组内分工合作，相互学习，真真正正的学到了东西，受益匪浅。

4 附录

附录一 删除 MySQL8.0

附录二 安装 MySQL5.7

附录三 分库分表配置文件

附录四 flask 源代码

附录五 solution 类源代码

附录六 MyDB 类源代码

附录一、二、三、四、五、六详情见踩坑+体会+源代码.md 文件。