跨数据源(catalog)的一些研究

由于组内项目面临大面积重构，初步的方案决定按业务重新划分数据源，将他们互相解耦出来；但由于有些页面报表，离线数据等需求是需要跨源去合并数据的，暂时的数据源只有Mysql，但不排除以后会使用ES、HIVE等其他源，所以在市场上开始找一些可以使用的高性能跨源组件。

通过查阅一些资料以及跟一些有类似经验的同学聊了之后是有了一些方案的：

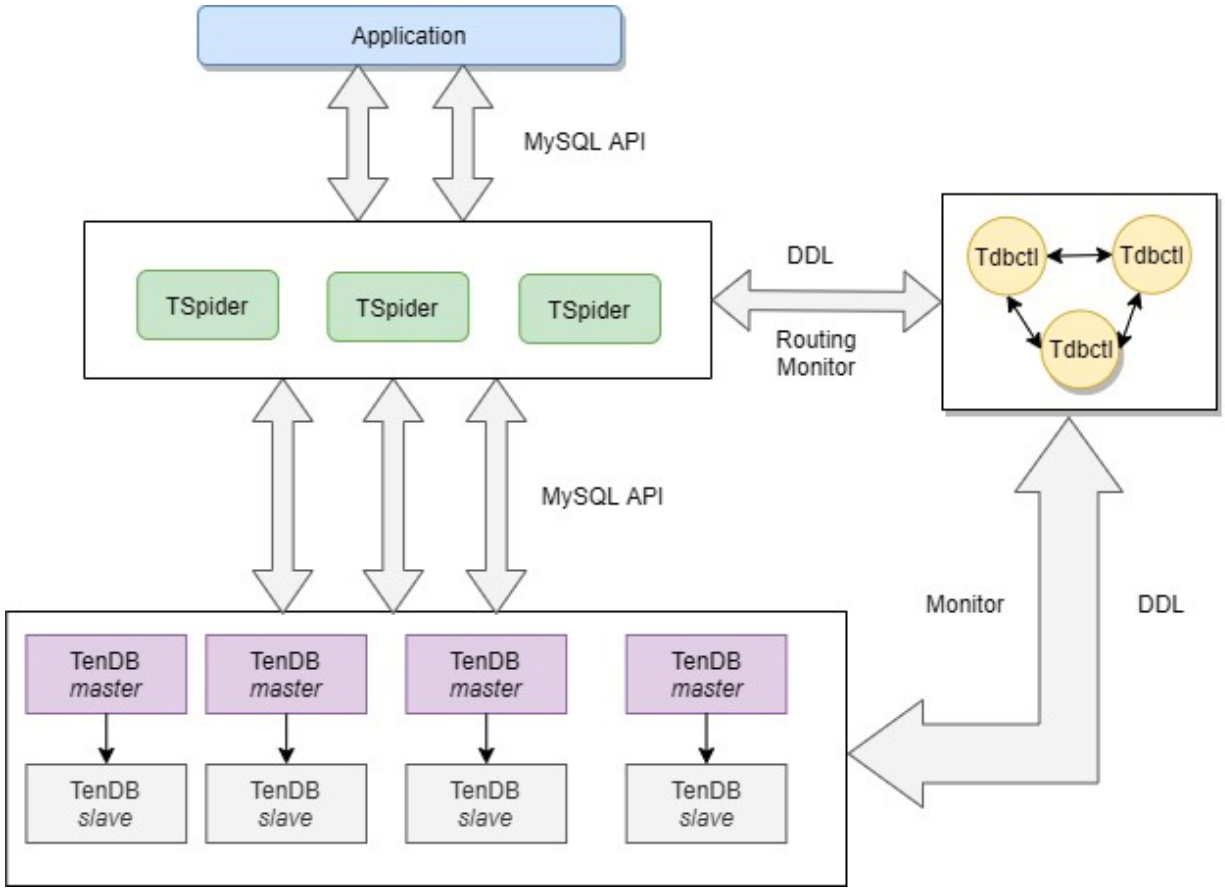
- 按需分库查出数据，然后在内存中用代码处理返回（内存占用大，可能导致频繁GC；并且性能无法得到保证，单机CPU性能有限；业务处理复杂，如果多表join，代码将变得难以维护）
- 使用Mysql的FEDERATED引擎来建立远程连接表（类似于Oracle的DBlink,但是只能跨Mysql数据源，并且性能也不是很高，对于远程表需要自己维护）
- 公司内部一些比较热的组件：Tspider（开源）、SuperSql（闭源）、PrestoDB（开源）前两者都是腾讯的组件，后面那个是Facebook开源的组件（还有个PrestoSql就是以前PrestoDB的团队单干的一个相似组件）

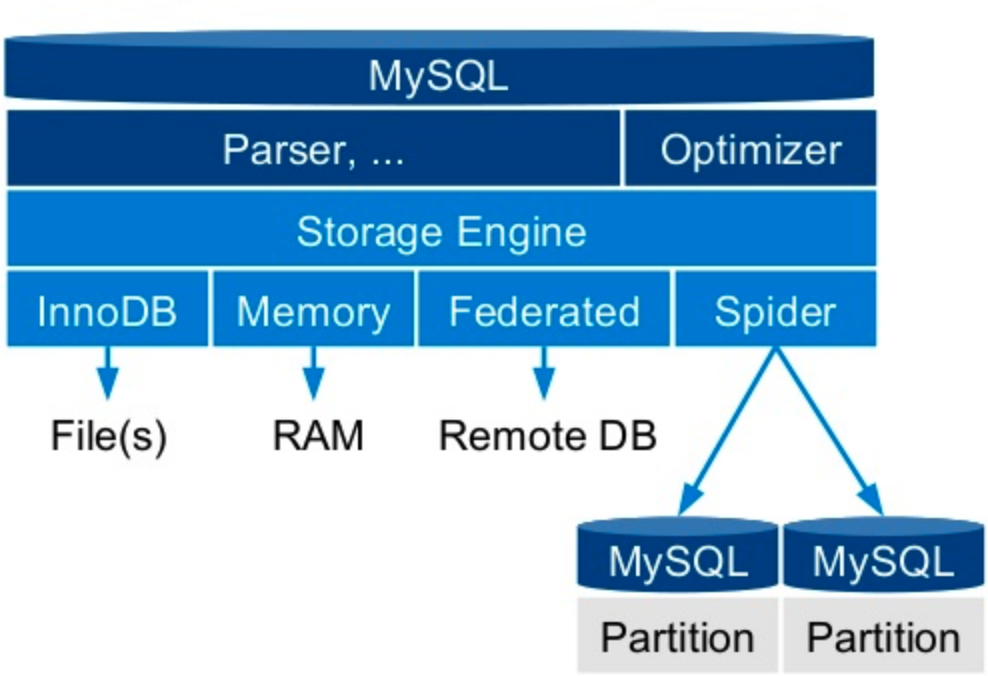
通过这些方面的对比，上述的那些组件是比较适合这次的技术选型；那么我们一起来看来上述三个组件的一些不同和优势：

- Tspider是分布式Mysql，透明分库分表、高可用的MySQL集群服务，透明及在线的扩容及缩容;TSpider也支持范围查询或跨分区查询及更新，但效率会差些，跨表查询的效率最差；所以它适用于点查询
- Presto使用场景主要是adhoc(即席查询：尽可能快的执行自定义SQL)，适合做探索性分析,跨数据源的功能不是很常用，但它支持了非常多的数据源，性能也有一定的保证；
- SuperSql尚未开源，就先看下这篇文章吧，有详细的介绍和性能测试；

Tspider

我们看一下来自Tspider官网的整体架构（这里得吐槽一下，得F12才能拿到page url）：


Application
MySQL API
TSpider
TSpider
TSpider
DDL
Routing
Monitor
Tdbctl
Tdbctl
Tdbctl
MySQL API
Monitor
DDL
TenDB
master
TenDB
master
TenDB
master
TenDB
master
TenDB
slave
TenDB
slave
TenDB
slave
TenDB
slave



上面官方文档介绍的很清楚，基本看一遍就能懂；

简单来讲就是TSpider由三大组件组成

- Tspider (TenDB Cluster集群的接入层)
- TenDB (TenDB Cluster的数据存储层)
- Tdbctl (TenDB Cluster集群的中央控制模块)

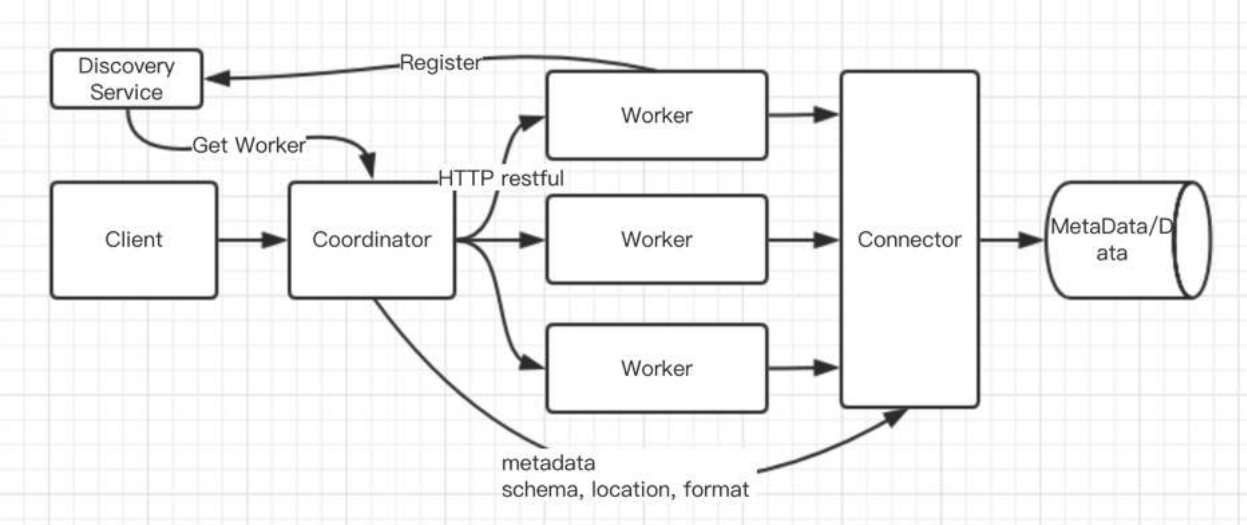
它参考了Mysql的分区表而做了一个Spider引擎，是基于开源框架Spider3.1改造升级的；但这个引擎不像分区表一样，它是不做数据存储的，是无状态的，它会根据分区配置将数据发送到TenDB上存储，如果是查询请求他也将去对应分区的TenDB上进行查询；那么这里可能就会产生一些问题，如果我们需要查询多个跨Partition的数据，那么最后的聚合效率肯定是要大打折扣的，就像我们分区表查询多个分区数据一样，引擎需要对多个分区进行查询，并且聚合返回，所以这也是它性能瓶颈的地方。看了下社区的发展路线，上面列举的点还是对SQL性能的优化，以及一些功能的扩展（贴近于Mysql的，比如全局唯一递增ID等)；

- 使用性方面，由于Tspider是基于Mysql开发的，所以他是后端语言无障碍的，可以使用Mysql的就可以
- 安全性方面，可以规避安全问题，全部交给Mysql来做即可

- 社区算不上活跃，但是公司内部使用较多，也可以得到更多实战的经验和教训；
- 业务支持方面，暂时只支持Mysql，其他数据源无法得到支持
- 性能方面，毕竟是个分布式Mysql组件，点查询给力，但是对于大数据量多分区join效率不高

Presto

前面介绍了下Presto用于即席查询较多，先看看它的Use Cases，然后可以来看看他的流程架构：



它的整个流程其实在上图已经画的很明确，其中Connector可以有多个也就是我们常说的Catalog，比如我们使用的Mysql、Hive等，它是一个基于内存的分布式查询引擎；

- 适用性方面，其中的Client可以直接使用Http请求代替，虽然官方提供了go、python等开源客户端；但是最简单的restful 的http请求也是可以支持的，这点还是比较好的，减少了对于客户端的依赖。
- 安全性上面，它提供了诸多的安全认证，最常用的我们可以在Presto Coordinator上对客户端做配置认证
- 社区方面PrestoDB在github上面是少有的star有11k的java项目，社区比较活跃
- 业务支持方面，Presto提供丰富的数据源的支持，并且对于数据源的性能也比传统的MMP框架有更高的性能，唯一美中不足的地方是，他只支持标准SQL，那么Mysql自

有的一些语法可能需要改造才能支持，比如说limit 、group by等，并且他对于跨源join的数据格式要求是一致的，否则需要进行cast

- 性能方面，除了官网的性能测试，我也自己进行了一些业务场景的测试（测试数据已脱敏），大致是秒级返回，对于并发数跟QPS大致是成线程成长趋势。

我前面介绍了怎么搭建Presto，贴一下这次测试的配置

Coordinator配置

etc/node.properties

```
node.environment=production
node.id=ffffffff-ffff-ffff-ffff-fffffffffff1
node.data-dir=xxx/data
```

etc/config.properties

```
coordinator=true
node-scheduler.include-coordinator=true#配置为既是Coordinator也是Worker
http-server.http.port=8080
query.max-memory=8GB
query.max-memory-per-node=2GB
query.max-total-memory-per-node=4GB
discovery-server.enabled=true
discovery.uri=http://xxx:8080
```

etc/jvm.config

```
-server
-Xmx6G
-XX:+UseG1GC
-XX:G1HeapRegionSize=32M
-XX:+UseGCOverheadLimit
-XX:+ExplicitGCInvokesConcurrent
-XX:+HeapDumpOnOutOfMemoryError
-XX:+ExitOnOutOfMemoryError
```

Worker配置

etc/node.properties

```
node.environment=production
node.id=ffffffff-ffff-ffff-ffff-fffffffffff2
node.data-dir=xxx/data
```

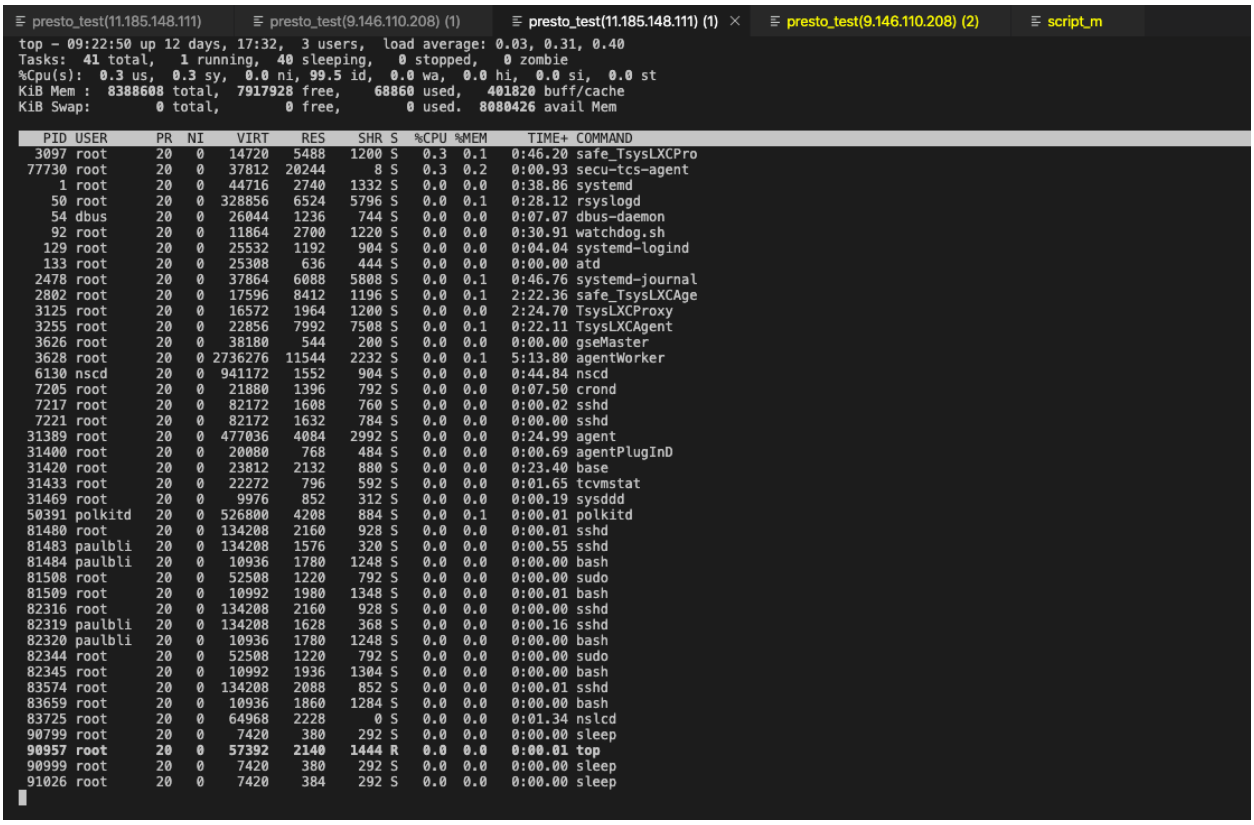
etc/config.properties

```
coordinator=false
http-server.http.port=8080
query.max-memory=8GB
query.max-memory-per-node=1GB
query.max-total-memory-per-node=2GB
discovery.uri=http://xxx:8080
```

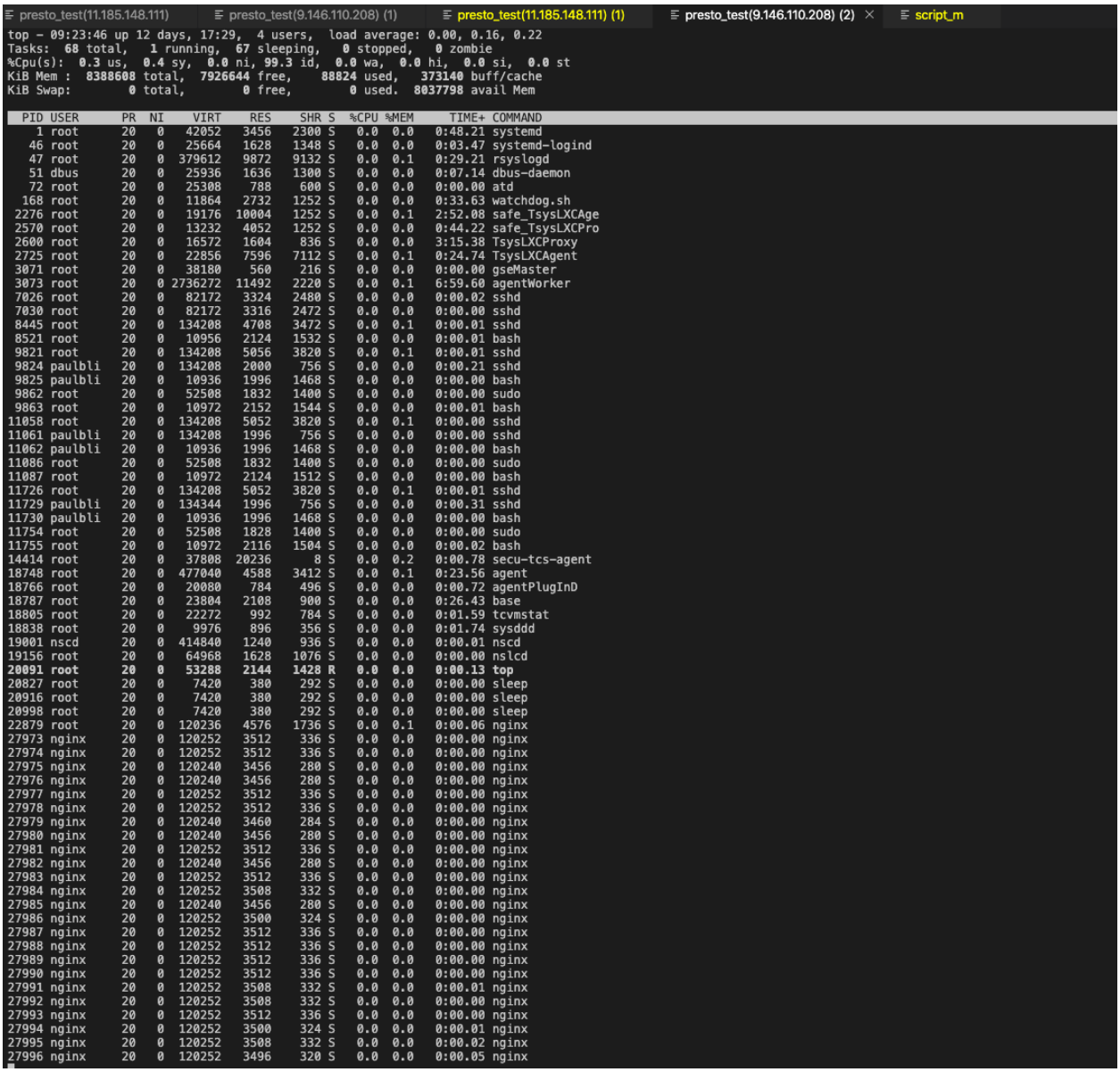
etc/jvm.config

```
-server
-Xmx6G
-XX:+UseG1GC
-XX:G1HeapRegionSize=32M
-XX:+UseGCOverheadLimit
-XX:+ExplicitGCInvokesConcurrent
-XX:+HeapDumpOnOutOfMemoryError
-XX:+ExitOnOutOfMemoryError
```

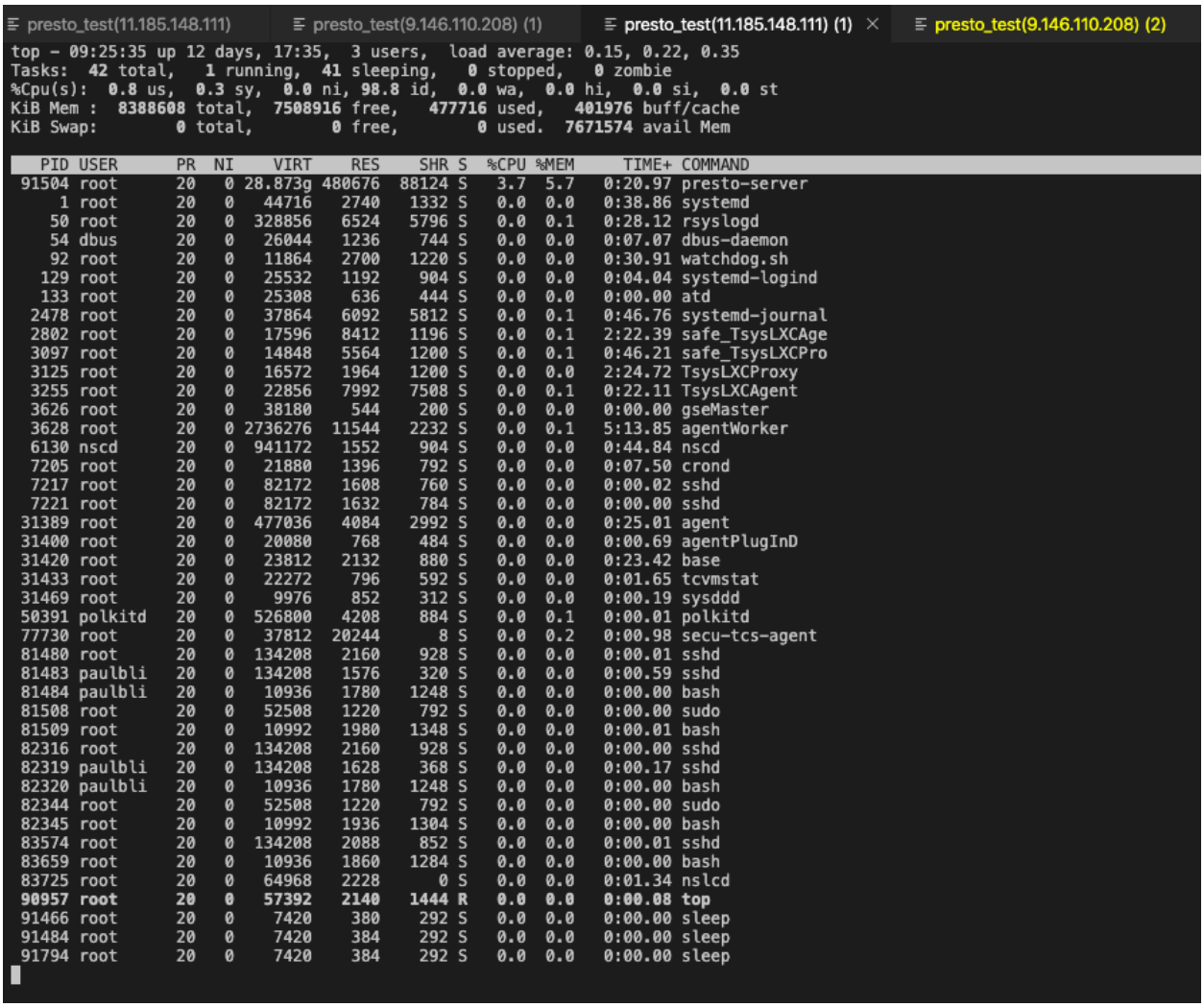
正常情况下coordinator机子的情况



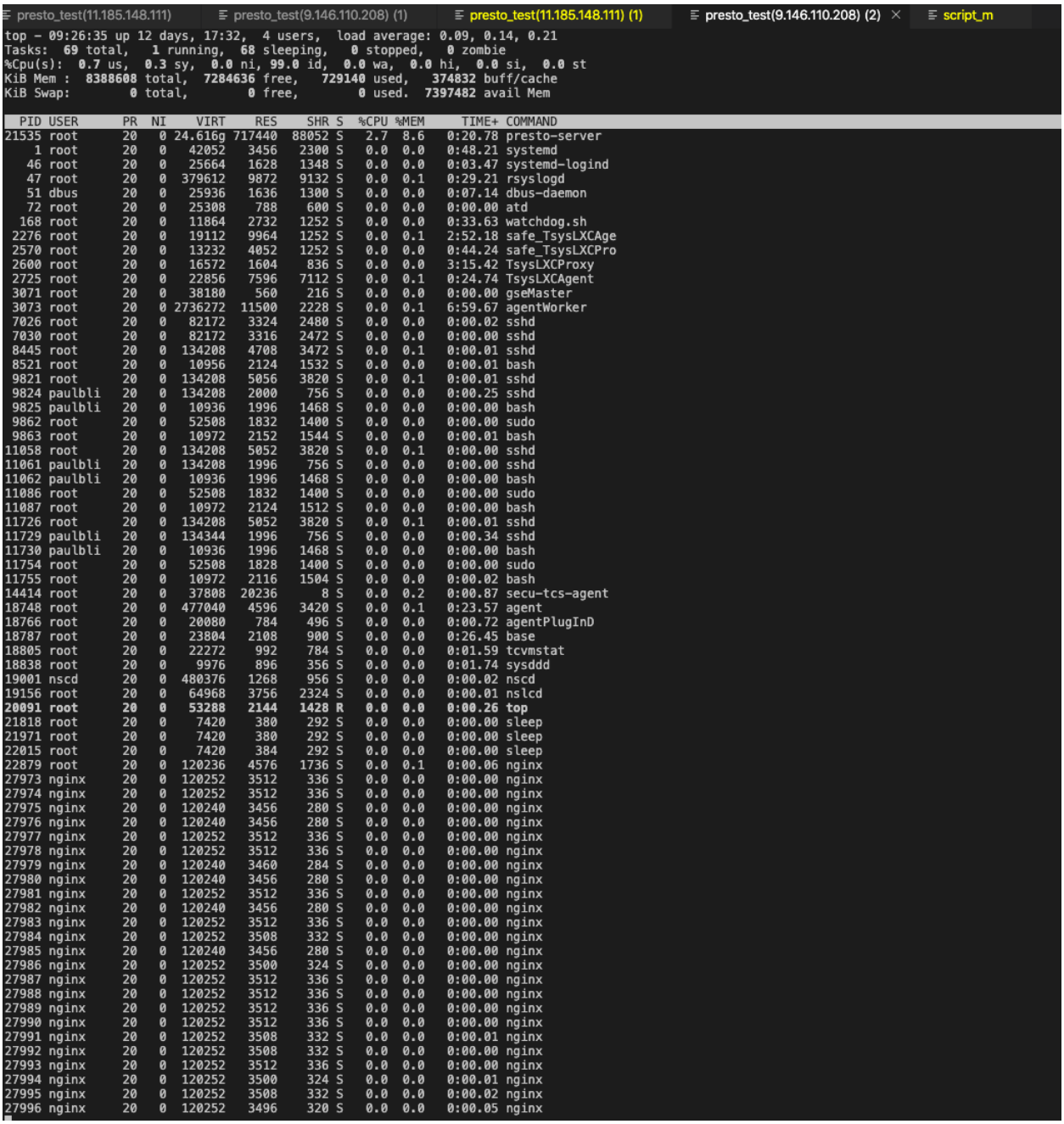
正常情况下worker机子的情况



启动presto-server之后的coordinator机子的情况



启动presto-server之后worker机子的情况



启动之后根据资源占用情况发现每台机子剩余内存不到7个g，所以对presto参数做了调整以保证他能拥有最大的使用率，由于coordinator 上也设置了启动worker，那么我们现在可用的就是这两个节点。

下面是分别用Python多线程和Python多进程的测试结果

先贴代码：

```
def run_thread(arg):
    print('thread ' + str(arg) + " running....")
    conn = prestodb.dbapi.connect(
        host='xxx',
        port=8080,
        user='xxx',
        catalog='xxx'
    )
    cur = conn.cursor()
    cur.execute("select * from xxx")
    begin = datetime.datetime.now()
    rows = cur.fetchall()
    end1 = datetime.datetime.now()
    time_interval = (end1 - begin).seconds
    print("producer time_interval:" + str(time_interval))
    print(len(rows))

if __name__ == '__main__':
    for i in range(10):
        run_thread('single'+str(i))
        # t = threading.Thread(target=run_thread, args=('threading'+str(i),))
        # t.start()
        # p = Process(target=run_thread, args=('process'+str(i),))
        # p.start()
```

由于资源有限，两台机子都是4核8G；分别对原生sql，优化后sql，使用单线程、多线程、多进程做了个比较：

未优化的sql花费时间

Name	5次循环	10次循环	20次循环
单线程	25	50	100
多线程	13~14	25~28	56~60
多进程	8~11	17~19	25~30

优化后的sql花费时间

Name	5次循环	10次循环	20次循环

Name	5次循环	10次循环	20次循环
单线程	10	20	40
多线程	8~9	16~18	28~30
多进程	6~7	11~13	16~19

总之从测试结果来看在资源耗尽的情况下多进程的效果仍旧是表现的最好的，但是在进程数到达一定数量后多线程的性能是会超过多进程的，因为这里只有IO请求没有任何计算，那么多线程的开销是要远远小于多进程的，我这里没法测试，因为我们Presto的服务器资源是有限的，过多的并发并不能反映最真实的结果。