Table of Contents

Cetus性能测试报告

编写人：何伟 版本 1.0.2 修改日期：2017.05.25

测试目的

本次测试主要是为了测试我们自身在MySQLProxy基础上改进开发的数据库中间件(以下简称cetus)在特定硬件环境下的性能，着重对比通过cetus访问与直连MySQL的性能对比情况，同时基于cetus的限制，以及后续可能的实际部署情况，我们也按照实际可能的部署方式进行了部署。验证了在实际部署情况下的性能参数，为后续系统的上线部署和相关性能参数评估提供了一定的依据。

测试工具

本次测试主要使用tpcc-mysql(以下简称tpcc)，其中，tpcc有两个版本，分别用于分片版本和非分片版本的测试，主要用来测试在模拟线上业务的情况下，cetus能支持到的业务情况。

测试环境

本次测试主要在杭州BGP机房的测试用服务器上进行，评估在系统真实线上应用时的性能数据,由于硬件条件的限制，本次性能测试在虚拟化环境上进行，数据库的单机CPU性能大概在实际机器的性能的三分之一左右，IO性能大概为实际环境的一多半（未做Raid1）。

硬件环境

Dell R730*2 Intel Xeon E5-2650 v3 @ 2.30GHz * 2(40 cpu core at all); 64G DDR3(Dual Channel); 800GSSD

Inspur SA5212M4*1 Intel Xeon E5-2630 v3 @ 2.40GHz * 2(40 cpu core at all); 128G DDR3(Dual Channel); 2 * 1T(raid 1)，2 * 800G SSD(raid 1)

Intel 2650v3 @ 2.30 GHz * 10， 12G Mem (KVM-tpcc)

Intel 2630v3 @2.4GHz*2 4GMem (KVM-cetus)

Intel 2650 @2.0GHz*2 4GMem(KVM-cetus)

Intel 2650v3 @ 2.30GHz * 8, 24G Mem, 800G SSD (KVM-mysql-server)

网络环境

测试环境网络

业务类型	IP地址	服务器类型	服务器操作系统
TPCC	host1	KVM	RHEL7.2
lvs	host2	Inspur SA5212M4	RHEL7.2
cetus	host3~host22	KVM	RHEL7.2
DB	host23~host26	KVM	RHEL6.8

测试各个服务器均在同一个机房，部分服务器不在同一个机柜上，可能会受到上联线路的影响。

20个cetus分布在两个物理服务器上，剩余资源较多的服务器上部署了15个cetus，权重为10，另一个服务器CPU性能较这个服务器有所差异、部署了5个cetus，权重为9。

各个网卡均是千兆全双工，线路质量较稳定。

软件环境

测试软件	版本	备注
MySQL	5.7.17	当前最新的5.7分支社区版本，仓库RPM安装
Cetus	6c72550f	5月9日最后提交版本
lvs	1.2.13-el7	红帽仓库版本
tpcc-mysql	1ec1c5eb@master	https://github.com/Percona-Lab/tpcc-mysql 禁用了server端的Prepare，最大重试次数修改为2，保证测试性能的可比性
tpcc-mysql-shard	bf1addd@wid	https://git.ms.netease.com/weihe/tpcc-mysql.git 基于wid进行分片，禁用server端prepare，修改最大重试次数为2。

软件配置

数据库使用2台R730物理服务器，使用KVM拆分为两个数据库，每个数据库有30G内存，为了测试分库版本和直连数据之间的数据差异，CPU限制为每虚拟机8个CPU，均分到对应的CPU的8个物理核心上。每个测试用的数据库都有独立的SSD挂载到虚拟机内，测试时IO带宽是独立的，基本不会相互影响。

MySQL采用了严格模式，每次提交都写入日志后再响应。

lvs，以及cetus的配置信息详见附录。

测试数据库相关信息

tpcc非分片数据库初始化信息

我们使用了tpcc自身的初始化工具进行了数据库的初始化。tpcc测试使用的是第一个数据库的上的非分片database，database名称为tpcc_noshard，本次测试中，我们在单一的数据库中初始化了200个warehouse,为了尽量保证测试程序的业务逻辑一致，我们去除了部分分片版本不支持的外键约束。

```
mysql -utpcc -pxxxxxx tpcc_noshard -hhost3 -P3360 <create_table.sql
bash ./load.sh tpcc_noshard 200 4

mysql -utpcc -pxxxxxx tpcc_noshard -hhost3 -P3360 <add_fkey_idx_noshard.sql
```

其中，我们是等待数据都加载完之后。再导入的索引和外键相关的依赖。

tpcc分片数据库初始化信息

我们使用了tpcc自身的初始化工具进行了数据库的初始化，初始化时类似业务操作，也是通过lvs进行的。tpcc测试使用的是四个数据库的上的相同的database，database名为tpcc_shard，本次测试中，我们新建了200个warehouse,为了尽量和分库版本的Cetus兼容，我们去除了部分分片版本不支持的外键约束。

```
mysql -utpcc -pxxxxxx tpcc_shard -hhost2 -P3360 <create_table.sql
bash ./load.sh tpcc_shard 200 4

mysql -utpcc -c -pxxxxxx tpcc -hhost2 -P3360 < shard_add_idx.sql
```

其中，我们是等待数据都加载完之后。再导入的索引和外键相关的依赖。

TPCC测试数据

测试数据介绍

测试场景

根据业务的需求，我们使用tpcc模拟经典的业务情境进行测试。

分库版本和直连mysql，都是200个warehouse。在每次测试前，会对mysql进行重启，将数据文件恢复至测试前，并清除文件读写cache，尽量保证每次测试的环境基本一致。

测试维度

本次测试我们主要考虑测试cetus在各个线程情况下的工作性能，最主要考察的是每分钟联机交易数，即TpmC，同时，我们也会分析5个环节中的延迟数据，对性能瓶颈进行分析。

为了对各个线程情况下的状态进行测试，同时，尽量测试出一个性能峰值，我们选取了从8个工作线程到3000个线程之间的多个点进行测试。

测试带有验证性质，每次测试没有执行太长时间。基本每个测试预热4分钟，评估30分钟。

待测试软件介绍

本次测试，我们主要关心部署了cetus之后，和直连myql直接的性能差异，同时，也通过lvs+cetus这种更接近线上部署的方式来确认lvs会给系统带来多大延迟，以及多个cetus之间的容量扩展是否线性。

详细测试数据

测试数据分为3大系列，分别为直连Mysql，单cetus，以及通过lvs做的cetus集群

Mysql	TpmC	New-Order	Payment	Order-Status	Delivery	Stock-Level
8	10354.8	31.9	8.9	4.4	80.3	125
16	19538.6	32.8	10.5	5	97	119.3
32	28598.699	43.6	16.1	6.8	153.4	137.5

32	28598.699	43.6	16.1	6.8	153.4	137.5
64	34040.301	72.2	29.4	10.6	276.5	200.4
128	35955	134.9	59.1	17.3	526.2	364.1
200	35590.699	211.5	97	24.9	829.6	556.2
256	35342.602	270.9	128.7	30.2	1075.1	694
300	35225.199	314.3	157.4	35.5	1275.7	785.2
400	34920	399.4	237.2	45.5	1815.1	933.8
512	34554.699	478.9	349.2	57.2	2550.8	958.8
768	33207.199	657.5	655.5	75.7	4508.5	763.2
1000	29792.301	730.9	1247.2	70	6433.6	424
1500	24433.199	1101.2	2556.1	79.2	10170.7	381.3
2000	26119.4	937.4	2049.7	74.5	8615	365.2
2500	24917.301	1038.1	2406.4	75.8	9610.4	368
3000	23110.5	1283.1	3002.5	83	11974.4	392.4
3000	23110.5	1283.1	3002.5	83	11974.4	392.4

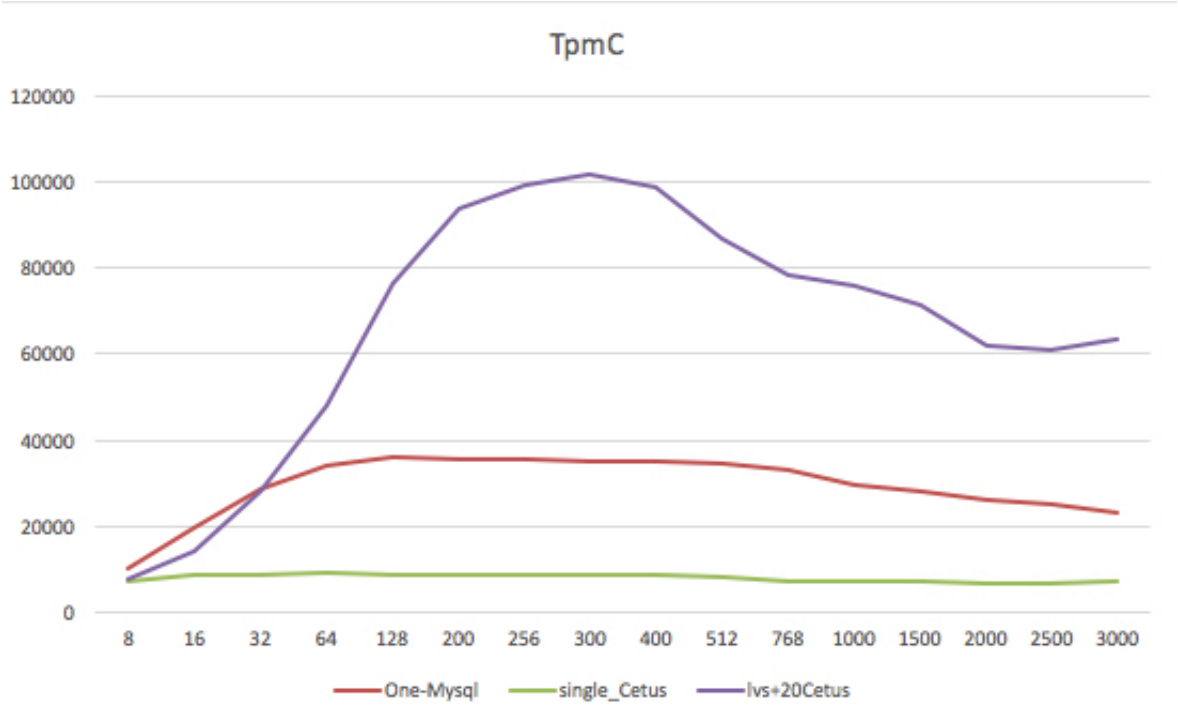
Single Cetus	TpmC	New-Order	Payment	Order-Status	Delivery	Stock-Level
8	7169.4	45.9	12.9	6.7	109.9	188.1
16	8762.8	75.6	20.3	10.7	173.4	317.8
32	8759.8	152.7	39.7	21.3	333.9	642.9
64	8917.4	300.9	79.2	42.6	641.9	1256.3
128	8678.3	618.9	166.2	84.6	1290.9	2575
200	8706.4	961.3	270.4	130.1	1978.1	3960
256	8711.6	1224	361.3	164.2	2493.9	5008.7
300	8562.3	1455.1	442.2	193.6	2941.6	5931.1
400	8516.3	1933.7	635	254.5	3858.6	7773.9
512	7966.1	2607.6	927.3	339.4	5136.4	10360.4
768	6926.3	4329.1	1679.8	555.3	8393.7	16941.9
1000	6972.9	5346.3	2196	683.6	10382	20967.7

1000	6972.9	5346.3	2196	883.6	10382	20967.7
1500	6972.1	6727.6	2866.3	862.2	13064.9	26449.7
2000	6859.4	7360	3116.9	949	14350.8	29035.8
2500	6895	7157.6	3027.2	920.5	13927.4	28167.3
3000	7136.9	5868.8	2480.2	751.3	11360.6	23006
LVS+20Cetus	TpmC	New-Order	Payment	Order-Status	Delivery	Stock-Level
8	7735.1	42.4	12	6.3	106.9	170
16	14283.8	45.5	12.7	6.3	120.4	186.7
32	27836	47.2	13.4	6.5	119.9	187.4
64	48235.801	54	17.2	8.3	135.2	206.2
128	76335.797	67.2	22	9.5	167.2	273
200	93998.5	83.9	28.2	10.7	205	364.6
256	99455	101	35	12.2	241.6	446.3
300	101627.203	115.4	41.8	14.2	272.8	503.4
400	98676	157.6	62	19.5	360.4	669.2
512	86811.797	225	104.5	27.2	485.6	913.9
768	78351.898	322.6	272.1	37.3	647.6	1198.3
1000	75835.898	379.9	448.3	42.7	728.4	1353.7
1500	71512.5	253.5	332.1	15.1	550.2	424.4
2000	61976.301	464.1	311.1	20.2	1029.1	457.4
2500	61085.102	795.1	411.7	25.3	2356.7	534
3000	63366.801	966.1	853.1	21.1	5689.1	434.3

数据对比图

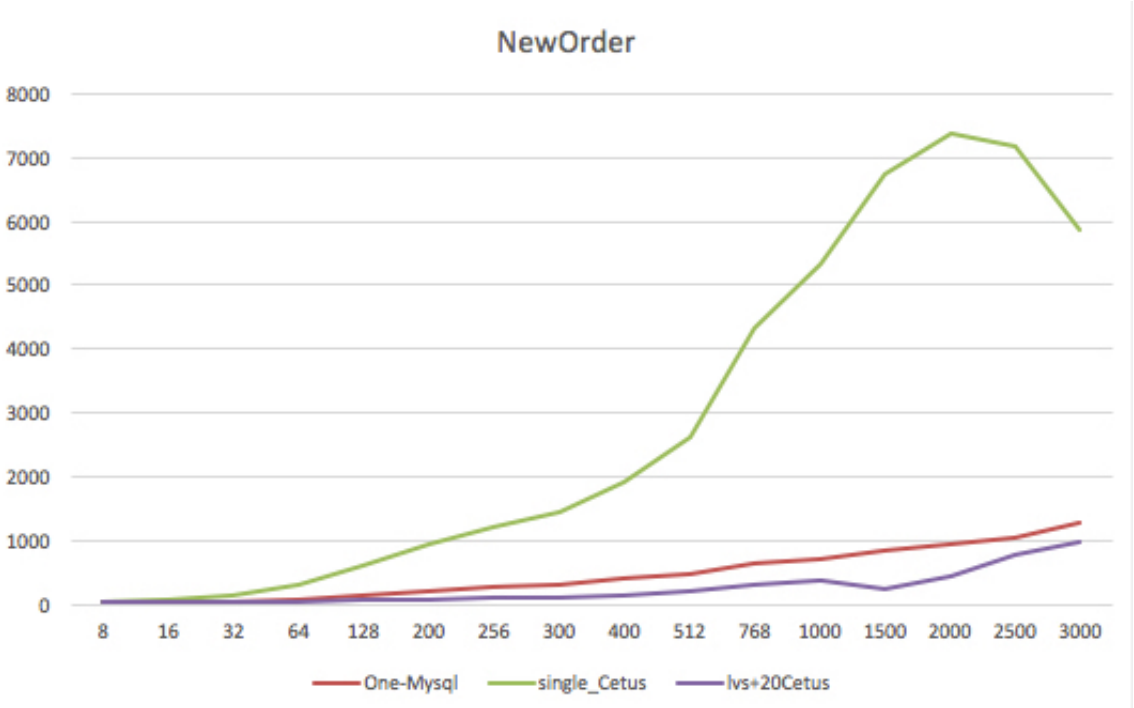
- TpmC对比图

  横坐标为测试线程数，纵坐标为TpmC(每分钟联机交易数)



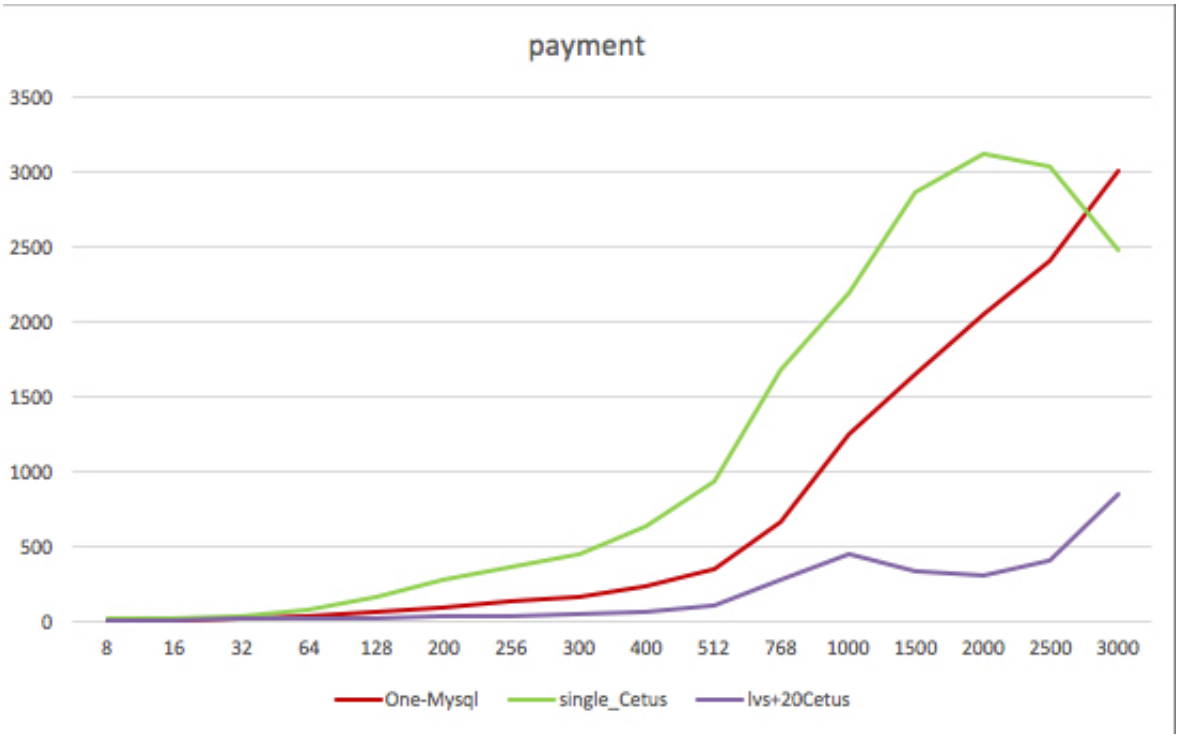
- NewOrder延迟对比图

  横坐标为测试线程数，纵坐标为新订单业务处理平均延迟，单位为毫秒



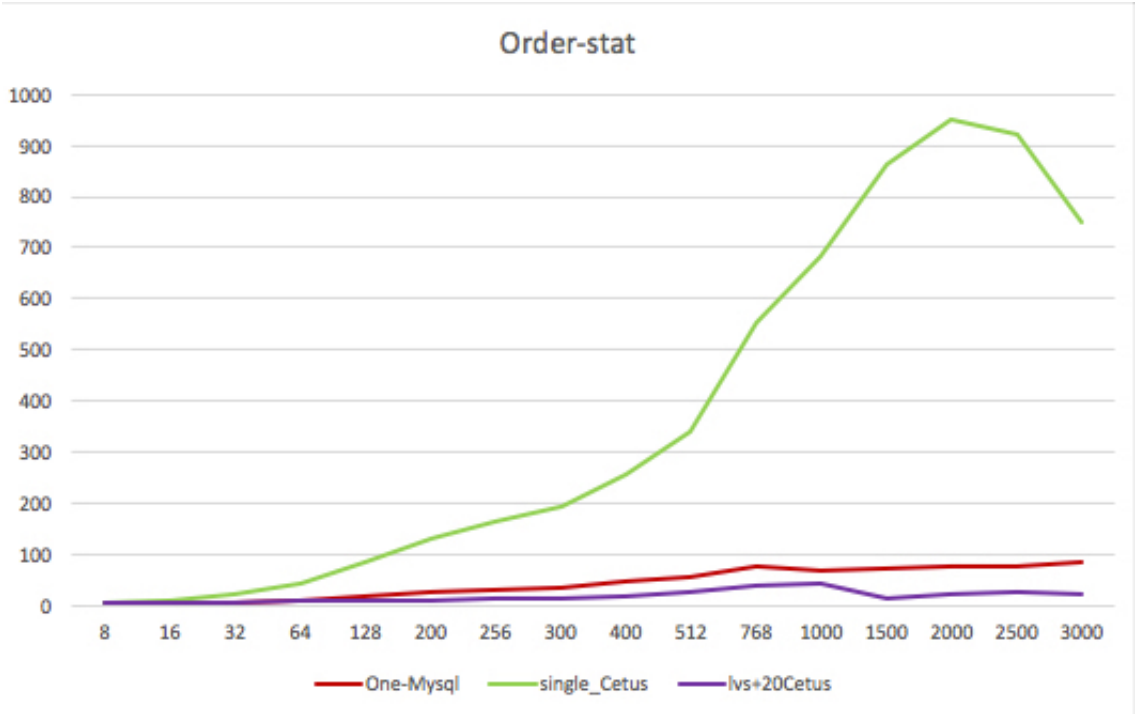
- payment时间对比图

  横坐标为测试线程数，纵坐标为支付业务处理平均延迟，单位为毫秒



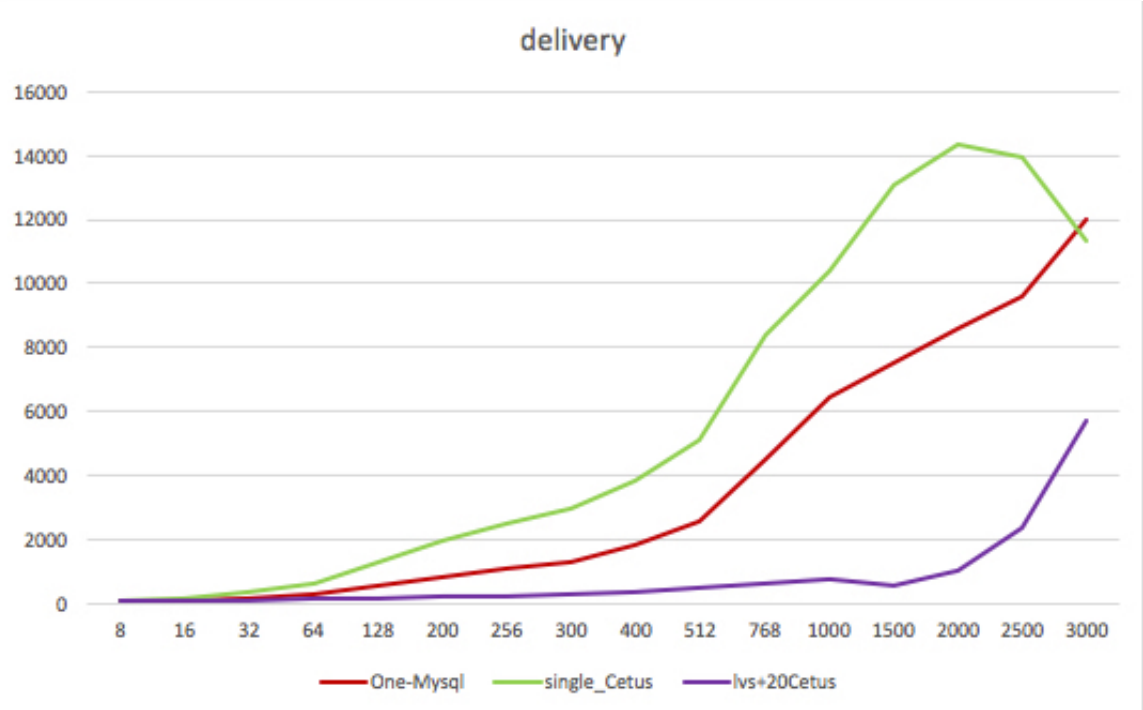
- OrderStat响应时间对比图

  横坐标为测试线程数，纵坐标为订单状态统计处理平均延迟，单位为毫秒



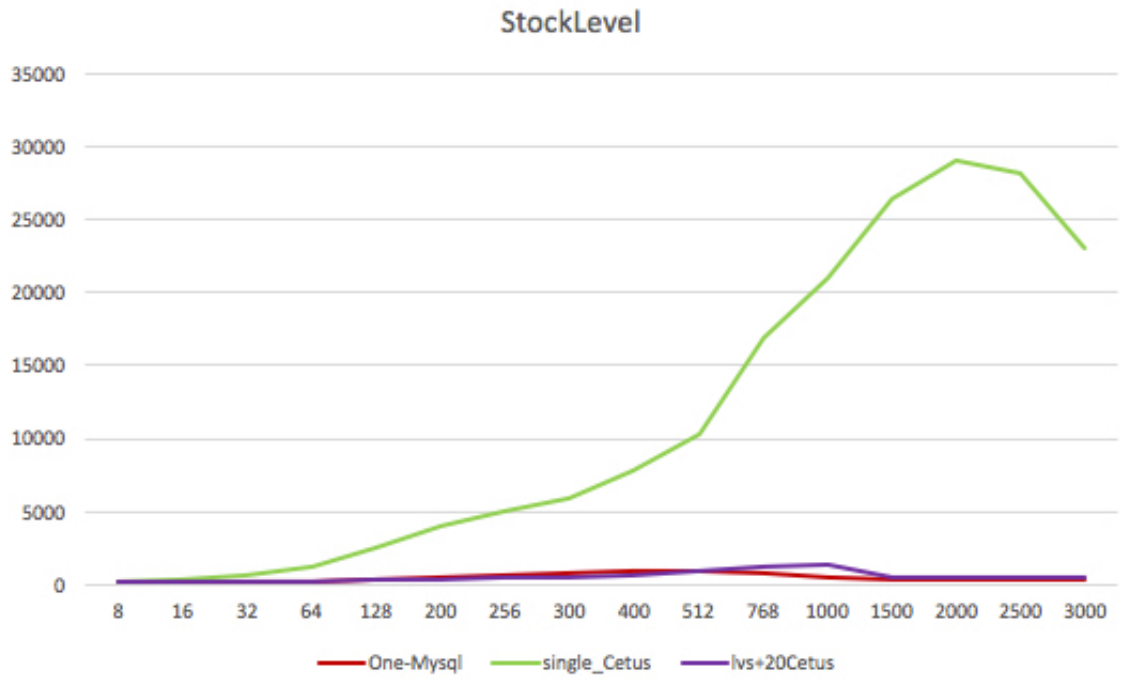
- Delivery响应时间对比图

  横坐标为测试线程数，纵坐标为订单发货业务处理平均延迟，单位为毫秒



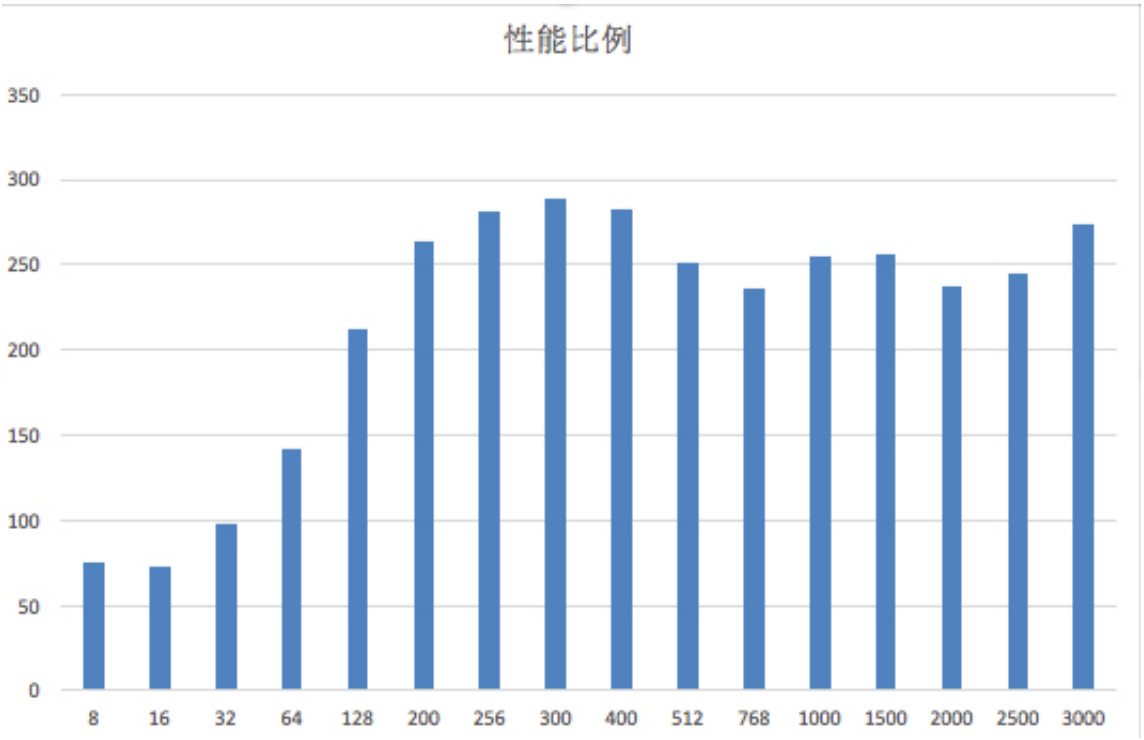
- StockLevel响应时间图

  横坐标为测试线程数，纵坐标为库存统计业务处理平均延迟，单位为毫秒



- TpmC性能比例图

  横坐标为测试线程数，纵坐标为分库版本的cetus与直连MySQL的性能百分比。



数据分析

单Cetus性能分析

单机处理能力

因为后端的数据库性能较好，且cetus在分片模式下，对SQL的解析，以及分布式事务的处理，也需要一定的开销，在8线程测试开始，cetus基本就处于最佳运行状态下，达到或接近了单机的最大处理TpmC，大约在7000左右，并一直稳定并保持到了3000连接。得益于我们最新的连接池机制，单cetus能够支持到3000+的前端连接。

单cetus在线测试环境下,由于数据库性能良好，cetus很快达到了自身的瓶颈，处理器CPU资源占用100%。随着连接数的增加，性能基本没有出现衰减。基于event的连接处理模型保证了处理速度不会随着监听连接的增长而快速下降。但是因为cpu处理能力的限制，会给请求带来额外的延迟，延迟会随着连接数的增加而显著增大，这个现象在分项延迟上可以很明显的看到趋势，最后在3000连接时性能小提升，可能是因为宿主机cpu睿频导致的单处理器性能提升导致的。

延迟分析

cetus会带来额外的延迟，额外延迟的表现体现在进行较少连接数测试时，cetus的TpmC显著低于直连MySQL，且延迟高于直连MySQL，根据不同事务的SQL数量和类型不同，延迟分别增加2毫秒到65毫秒。随着并发的增加，CPU资源有限，单个cetus自身带来的延迟显著增加。

多Cetus集群部署

多cetus集群部署，采用lvs做前端。便于进行在线的业务维护。同时lvs也提供了故障迁移，负载均衡的功能，为软件的平稳运行提供了很好的条件,测试时，我们根据Cetus所在机器的性能差异，分别设置了不同的权重。

Cetus的集群扩展性

在我们进行测试时，使用的是20个cetus并行提供服务。通过我们的吞吐量对比测试，20个cetus的处理性能，在200线程到2000线程的范围内，性能是单个cetus的10倍左右。在3000线程数时，性能仍有单个cetus的8倍，水平扩展能力较好。

分库带来的性能提升

我们测试时，实际上有4个mysql实例，采用了基于hash的分库方式，每个分库里有50个warehouse的数据。测试中可能会因为跨warehouse的事务带来XA事务，后续我们除了整体TpmC，也会有针对性的针对XA事务以及单机事务带来的性能差异进行分析。

整体性能分析

为了模拟硬件瓶颈，我们搭建了4个硬件配置一致的MySQLServer，直连MySQL的数据是连接其中的一个库测试的。测试期间，其他的库无任务。分库测试时，我们通过lvs分发请求到Cetus上，cetus根据请求内容，智能分发数据到后端的数据库实例。从性能对比图来比较的话，在小于等于32工作线程时，由于cetus自身解析SQL，以及增加的数据包网络转发延迟，导致实际的性能比单机性能更差。大约在单机实际性能的75%到100%之间。当然，当前的业务压力，单MySQL的性能未达到瓶颈，可以完全满足测试需求，但在64线程及后续的测试中，由于单数据库自身计算资源有限，TpmC等迅速到达峰值并基本稳定。而分库版本的测试，在cetus的支持以及后端的多个硬件的合作下，性能超过了单机性能，并有较大的提升。在200~1500连接之间，都能保证至少2.5倍的性能数据。在128线程及以上的工作压力下，均能保证两倍以上TpmC数据，性能提升明显。

性能小结

分片版本Cetus在连接数少，业务繁忙环境下，因为网络延迟以及自身处理带来的额外延迟，性能大约为直连MySQL的75%左右，这种延迟因为每个连接上多业务顺序无等待排队执行，延迟被不断累积，表现为测试数据较差，根据我们的实际业务模型，多连接、每连接上任务数较为零散，实际运行时的整体性能的差异应该会比测试数据更小，应该能达到或接近直连MySQL的水平。在多连接、高负载的测试环境下，物理上的分库，极大的提升了系统的总体容量，在使用4倍硬件资源的情况下，能得到大约2.8倍的性能表现。

分项延迟分析

单cetus延迟分析

单cetus因为自身cpu资源限制，带来的等待延迟很高。其延迟数据不能代表后端处理延迟。

直连MySQL与Cetus集群延迟分析

延迟分析，我们分为两个部分进行。一部分是neworder和payment，这两个处理流程可能涉及分布式事务。另外是其他3个处理流程根据业务模型，我们可以走单机事务。我们针对这两种情况分项来进行延迟比较。

neworder中，XA事务大概占比1%，payment中，XA事务占比约15%。从图中，我们可以看到，随着线程数的增加，延迟增长最快的是payment。因为payment中涉及到一处频繁修改的热点数据，需要频繁的等待锁。

我们针对直连MySQL以及通过Cetus集群连接，整理了部分延迟数据对比：

这里列的是在3000线程数时，总的tpmc以及分项延迟与8线程时的一个数据对比。8线程时，因为数据库负载较轻，我们认为延迟是较为理想的，并且，以各项数据在8线程时的延时数据为基准，计算了在3000工作线程时，各项工作的实际延迟。

TPCC	TpmC	New-Order	Payment	Order-Status	Delivery	Stock-Level
MySQL	2.23	40.22	337.36	18.86	149.12	3.14
Cetus	8.19	22.79	71.09	3.35	53.22	2.55

从延迟数据可以看到。在负载增加的情况下，采用cetus集群加多个数据库的解决方案，能较好的控制业务延迟的增长。在单库的情况下，因为总体负载增加，锁冲突增加，payment的延迟增加非常的严重。在3000线程时的延迟是8线程时的340倍，影响了整体性能的提升。

而采用cetus集群的测试结果来看，虽然延迟有上升，但是根据业务的不同，上升的幅度比较有限。Order-Stat和Stock-level这两项单机事务的延迟上升非常轻微，发货业务（Delivery）由于整个流程涉及到了10个分开的子事务，且每个子事务均有数个update操作以及一个统计求和的操作，所以整体延迟增加较多。New-Order和Payment的延迟增加倍数，和XA事务也有一定的关系。XA事务需要cetus花费更多的时间去参与XA事务协调。需要等待所有节点返回Prepare成功之后才能提交并返回给用户。在实际的的使用中，我们需要尽量控制XA事务的比例，避免在高负载的情况下，性能恶化速度加快。

一句话总结

Cetus集群能在单库硬件资源达到或者快达到瓶颈的情况下，有效的提升系统的总体吞吐量，并显著的降低业务延迟。

附录：

软件的配置信息

Cetus(with lvs)

cetus的进程每个进程的启动都有点小的不同。我们以host3的配置文件为例，他的对应修改下IP信息即可。

```
[mysql-proxy]
daemon=true
plugins=shard,admin
proxy-address=0.0.0.0:3360
admin-address=host3:4360
admin-username=proxyadmin
admin-password=passwd
admin-lua-script=./lib/mysql-proxy/lua/admin.lua
plugin-dir=./lib/mysql-proxy/plugins
pid-file = ./logs/mysql-proxy.pid
log-file = ./logs/mysql-proxy.log
shard-part-file=./conf/part.json

proxy-backend-
addresses=host23:3360@data1,host24:3360@data2,host25:3360@data3,host26:3360
@data4

default-db=tpcc_shard
default-username=tpcc
log-level=debug
user-pwd=tpcc@xxxxxx
app-user-pwd=tpcc@xxxxxx
default-pool-size=200
max-pool-size=500
disable-threads=true


#proxy_debug_options
verbose-shutdown=true
log-backtrace-on-crash=true
```

分片信息配置

cetus同时还需要配置分片信息，从上文的配置文件中，我们可以看到，分片信息储存在part.json中，该配置文件内容如下：

```
{
  "vdb": [
     {
      "id":1,
      "type":"int",
      "method":"hash",
      "num":4,
      "partitions":{"data1":[0], "data2":[1], "data3":[2], "data4":[3]}
     }
  ],
  "table": [
    {"vdb": 1, "db": "tpcc_shard", "table": "warehouse", "pkey": "w_id"},
    {"vdb": 1, "db": "tpcc_shard", "table": "district", "pkey": "d_w_id"},
    {"vdb": 1, "db": "tpcc_shard", "table": "customer", "pkey": "c_w_id"},
    {"vdb": 1, "db": "tpcc_shard", "table": "new_orders", "pkey":
"no_w_id"},
    {"vdb": 1, "db": "tpcc_shard", "table": "orders", "pkey": "o_w_id"},
    {"vdb": 1, "db": "tpcc_shard", "table": "order_line", "pkey":
"ol_w_id"},
    {"vdb": 1, "db": "tpcc_shard", "table": "stock", "pkey": "s_w_id"},
    {"vdb": 1, "db": "tpcc_shard", "table": "history", "pkey": "h_c_w_id"}
  ],
  "default": {
    "tpcc_shard": "data4"
  },
  "all": {
    "tpcc_shard": ["data1", "data2", "data3", "data4"]
  }
}
```

keepalived配置

我们需要keepalived进行配置。保证我们可以通过统一的端口访问后面的4个服务。

```
! Configuration File for keepalived

global_defs {
   notification_email {
     weihe@corp.netease.com
   }
   notification_email_from keepalived@hostkp
   smtp_server 127.0.0.1
   smtp_connect_timeout 30
   router_id rd200
}

vrrp_instance VI_1 {
   state MASTER
    interface br0
```

```
    virtual_router_id 200
    priority 200
    advert_int 1
    authentication {
        auth_type PASS
        auth_pass 1111243
    }
    virtual_ipaddress {
        host2/22 dev br0
    }
}

virtual_server host2 3360 {
    delay_loop 6
    lb_algo wrr
    lb_kind DR
    persistence_timeout 0
    protocol TCP

    real_server host3 3360 {
        weight 10
        TCP_CHECK {
            connect_ip host3
            connect_port 3360
            connect_timeout 3
            nb_get_retry 3
            delay_before_retry 3
        }
    }
#从host3~host17的weight都是10
#省略了重复的18个rs的配置
#从host18~host22的weight是9
    real_server host22 3360 {
        weight 9
        TCP_CHECK {
            connect_ip host22
            connect_port 3360
            connect_timeout 3
            nb_get_retry 3
            delay_before_retry 3
        }
    }
}
```

tpcc初始化索引的文件

shard_add_idx.sql

```
CREATE /*# group="all" */ INDEX idx_customer ON customer
(c_w_id,d_id,c_last,c_first);
CREATE /*# group="all" */ INDEX idx_orders ON orders
(o_w_id,d_id,o_c_id,o_id);
CREATE /*# group="all" */ INDEX fkey_stock_2 ON stock (s_i_id);
CREATE /*# group="all" */ INDEX fkey_order_line_2 ON order_line
(ol_supply_w_id,ol_i_id);

CREATE /*# group="all" */ INDEX fkey_history_1  ON history
(h_c_w_id,h_c_d_id,h_c_id) ;
CREATE /*# group="all" */ INDEX fkey_history_2  ON history  (h_w_id,d_id) ;
CREATE /*# group="all" */ INDEX fkey_order_line_2  ON order_line
(ol_supply_w_id,ol_i_id) ;
CREATE /*# group="all" */ INDEX fkey_stock_2  ON stock (s_i_id) ;
```

tpcc非分区方式初始化索引文件

add_fkey_idx_noshard.sql

```
SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0;
SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0;


CREATE INDEX idx_customer ON customer (c_w_id,c_d_id,c_last,c_first);
CREATE INDEX idx_orders ON orders (o_w_id,o_d_id,o_c_id,o_id);
CREATE INDEX fkey_stock_2 ON stock (s_i_id);
CREATE INDEX fkey_order_line_2 ON order_line (ol_supply_w_id,ol_i_id);

CREATE /*# group="all" */ INDEX fkey_history_1  ON history
(h_c_w_id,h_c_d_id,h_c_id) ;
CREATE /*# group="all" */ INDEX fkey_history_2  ON history  (h_w_id,h_d_id)
;
CREATE /*# group="all" */ INDEX fkey_order_line_2  ON order_line
(ol_supply_w_id,ol_i_id) ;
CREATE /*# group="all" */ INDEX fkey_stock_2  ON stock (s_i_id) ;


SET FOREIGN_KEY_CHECKS=@OLD_FOREIGN_KEY_CHECKS;
SET UNIQUE_CHECKS=@OLD_UNIQUE_CHECKS;
```