基于 MyBatis 的关联实现方案

摘　要:　本文介绍了在基于 MyBatis 的 SpringBoot 应用中实现数据表关联的三种不同方案，并通过性能测试工具 JMeter 对这三种方案进行了速度差异和服务器负载的比较。实验结果显示，使用 JOIN 查询的方案在高并发情况下具有更好的响应时间和稳定性，为优化数据库查询提供了有效的实践指导。

关键词:　MyBatis；SpringBoot；数据表关联；JOIN 查询；性能测试；JMeter

问题描述：

在现代 Web 应用程序开发中，数据库操作是不可或缺的一部分，而数据表之间的关联查询更是常用的功能之一。然而，不同的关联实现方式对系统的性能有着显著的影响。本实验旨在探讨并比较在基于 MyBatis 框架的 SpringBoot 应用中，使用 ResultMap 关联、Dao 层单独查询以及 JOIN 查询三种方法实现数据表关联的效率。实验将通过构建特定的应用场景——查询产品完整信息，来评估这三种方法在不同并发量下的表现。

实验设计：

1. 实验环境

- 服务器 A: Ubuntu 18.04 服务器 2 核 1G 内存虚拟机一台，安装 docker，Maven、git，作为管理机，用于编译 productdemoaop 服务器 B：Ubuntu 18.04，2 核 1G 内存，运行 Docker，部署 MySQL 数据库
- 服务器 B：Ubuntu 18.04 服务器 2 核 1G 内存虚拟机一台，安装 docker，部署 productdemoaop Docker
- 服务器 C：Ubuntu 18.04 服务器 2 核 1G 内存虚拟机一台，安装 docker，部署 MySQL Docker
- 服务器 D：Ubuntu 18.04 服务器 2 核 1G 内存虚拟机一台，安装 JMeter 5.6.3,用于测试

2. 实验步骤

实验内容及要求如下：

在基于 MyBatis 的 SpringBoot 应用中，表的关联可以用三个方案完成

1. 利用 MyBatis 的 ResultMap 的关联来时实现对象的关联

2. 用 MyBatis 中单独查询一个对象，在 Dao 层来实现对象的关联

3. 在 MyBatis 中做一个一个 join 查询，用 join 查询的结果来实现对象的关联

在 productdemoaop 中，以查询产品完整信息的链接例子,比较上述三种方案的速度差异和服务器的负载。

其中方案 1 和方案 2 在 productdemoaop 中已经实现，方案 3 需要自己写代码实现。

类型	API 描述链接
Dao 层关联	GET /customer/products?name=xxxx&type=auto
MyBatis 关联	GET /customer/products?name=xxxx&type=manual

实验过程中除了需要用 Jmeter 测试三个方案的速度差异，也需要监控服务器 B 和 C 的负载情况。可以使用华为云云监控服务监控服务器的负载情况。

(1) ResultMap 的关联和 Dao 层关联两个方案在 productdemoaop 中已经实现，下面仅仅分析 Jmeter 测试的两个方案与新增方案的速度差异

(2) 方案 1 和方案 2 都需要查询三次数据库，方案 3 通过自己写代码，实现**只查询一次数据库**来获得想要的数据

方案 3 代码实现过程如下：

Ⅰ.前端会请求到 controller 中的如下方法，并且会根据 API 描述链接结尾**"type=?"**的内容判断调用哪种查询方法。type=auto 时请求的是 Dao 层关联，type=manual 时请求的是 MyBatis 关联，此处新增了一个 **type=join** 实现新增的方案 3

```
@GetMapping("")
public ReturnObject searchProductByName(@RequestParam String name, @RequestParam(required = false, defaultValue = "auto") String type) {
    ReturnObject retObj = null;
    List<Product> productList = null;

    if (null != type && type.equals("manual")){
        productList = productService.findProductByName_manual(name);
    } else if (null != type && type.equals("join")) {
        productList = productService.findProductByName_joinManual(name);
    } else {
        productList = productService.retrieveProductByName(name, all: true);
    }

    List<ProductDto> data = productList.stream().map(o->CloneFactory.copy(new ProductDto(),o)).collect(Collectors.toList());
    retObj = new ReturnObject(data);
    return retObj;
}
```

Ⅱ.调用 type=join 的 API 时，会调用 service 层新增的 **findProductByName_joinManual** 方法，然后 service 层调用 dao 层的 **findProductByName_joinManual** 方法。

```
public List<Product> findProductByName_joinManual(String name) {
    return productDao.findProductByName_joinManual(name);
}
```

Ⅲ.由于方案 3 要实现通过 join 连接查询只访问一次数据库查询到需要的数据，下面从 mapper 层开始分析。

我们需要先通过前端传来的 String 类型变量 name 查询 goods_product 表中的数据，然后通过数据里的主键 id 作为 product_id 查询 goods_onsale 表中的数据，再通过 goods_id 在 goods_product 表中查询 goods_id 相同的数据。

理清了查询方案后，可以写一个连接查询，一次性将所需数据都查询出来。**但是原有模块中并没有对应的类可以同时接受两张表的数据**，于是这里新建一个 ProductJoinPo 类接收查询结果，并在 mapper 层编写对应的 SQL 语句

```java
public class ProductJoinPo {
    // ProductPo 的属性
    private Long id; // goods_product.id
    private Long shopId; // goods_product.shop_id
    private Long goodsId; // goods_product.goods_id
    private Long categoryId; // goods_product.category_id
    private Long templateId; // goods_product.template_id
    private String skuSn; // goods_product.sku_sn
    private String name; // goods_product.name
    private Long originalPrice; // goods_product.original_price
    private Long weight; // goods_product.weight
    private String barcode; // goods_product.barcode
    private String unit; // goods_product.unit
    private String originPlace; // goods_product.origin_place
    private Long creatorId; // goods_product.creator_id
    private String creatorName; // goods_product.creator_name
    private Long modifierId; // goods_product.modifier_id
    private String modifierName; // goods_product.modifier_name
    private LocalDateTime gmtCreate; // goods_product.gmt_create
    private LocalDateTime gmtModified; // goods_product.gmt_modified
    private Byte status; // goods_product.status
    private Integer commissionRatio; // goods_product.commission_ratio
    private Long shopLogisticId; // goods_product.shop_logistic_id
    private Long freeThreshold; // goods_product.free_threshold
```

```
// OnsalePo 的属性
private Long onSaleId; // goods_onsale.id
private Long productId; // goods_onsale.product_id
private Long price; // goods_onsale.price
private LocalDateTime beginTime; // goods_onsale.begin_time
private LocalDateTime endTime; // goods_onsale.end_time
private Integer quantity; // goods_onsale.quantity
private Byte type; // goods_onsale.type
private Long onSaleCreatorId; // goods_onsale.creator_id
private String onSaleCreatorName; // goods_onsale.creator_name
private Long onSaleModifierId; // goods_onsale.modifier_id
private String onSaleModifierName; // goods_onsale.modifier_name
private LocalDateTime onSaleGmtCreate; // goods_onsale.gmt_create
private LocalDateTime onSaleGmtModified; // goods_onsale.gmt_modified
private Integer maxQuantity; // goods_onsale.max_quantity
private Byte invalid; // goods_onsale.invalid
```

```
@Select("""
SELECT p.id, p.shop_id AS shopId, p.goods_id AS goodsId, p.category_id AS categoryId,
       p.template_id AS templateId, p.sku_sn AS skuSn, p.name, p.original_price AS originalPrice,
       p.weight, p.barcode, p.unit, p.origin_place AS originPlace, p.creator_id AS creatorId,
       p.creator_name AS creatorName, p.modifier_id AS modifierId, p.modifier_name AS modifierName,
       p.gmt_create AS gmtCreate, p.gmt_modified AS gmtModified, p.status,
       p.commission_ratio AS commissionRatio, p.shop_logistic_id AS shopLogisticId,
       p.free_threshold AS freeThreshold, o.id AS onSaleId, o.product_id AS productId,
       o.price, o.begin_time AS beginTime, o.end_time AS endTime, o.quantity, o.type,
       o.creator_id AS onSaleCreatorId, o.creator_name AS onSaleCreatorName,
       o.modifier_id AS onSaleModifierId, o.modifier_name AS onSaleModifierName,
       o.gmt_create AS onSaleGmtCreate, o.gmt_modified AS onSaleGmtModified,
       o.max_quantity AS maxQuantity, o.invalid
FROM goods_product p
LEFT JOIN goods_onsale o ON p.id = o.product_id
WHERE p.name = #{name} OR p.goods_id IN (SELECT goods_id FROM goods_product WHERE name = #{name})
""")
List<ProductJoinPo> selectProductJoinByName(@Param("name") String name);
```

Ⅳ.和 auto、manual 查询方法一样，我们不能直接将 ProductJoinPo 类进行业务处理再传回前端，需要使用 CloneFactory 中的方法进行类的类型转换，因此在 CloneFactory 类中新增以下**静态方法**

将 ProductJoinPo 的 Onsale 属性提取出来

```java
/**
 * 将 ProductJoinPo 的 OnSale 属性提取出来
 * @param target
 * @param source
 * @return
 */
public static OnSale copy(OnSale target, ProductJoinPo source){
    // 创建 OnSale 对象并填充其属性
    target = OnSale.builder()
            .id(source.getOnSaleId())
            .price(source.getPrice())
            .beginTime(source.getBeginTime())
            .endTime(source.getEndTime())
            .quantity(source.getQuantity())
            .maxQuantity(source.getMaxQuantity())
            .creator(User.builder()
                    .id(source.getOnSaleCreatorId())
                    .name(source.getOnSaleCreatorName())
                    .build())
            .modifier(User.builder()
                    .id(source.getOnSaleModifierId())
                    .name(source.getOnSaleModifierName())
                    .build())
            .gmtCreate(source.getOnSaleGmtCreate())
            .gmtModified(source.getOnSaleGmtModified())
            .build();
    return target;
}
```

将 ProductJoinPo 的 Product 属性提取出来

```java
/**
 * 将 ProductJoinPo 的 Product 属性提取出来
 * @param target
 * @param source
 * @return
 */
public static Product copy(Product target, ProductJoinPo source){
    // 创建 Product 对象并填充其属性
    target = Product.builder()
            .id(source.getId())
            .skuSn(source.getSkuSn())
            .name(source.getName())
            .originalPrice(source.getOriginalPrice())
            .weight(source.getWeight())
            .barcode(source.getBarcode())
            .unit(source.getUnit())
            .originPlace(source.getOriginPlace())
            .commissionRatio(source.getCommissionRatio())
            .freeThreshold(source.getFreeThreshold())
            .creator(User.builder()
                    .id(source.getCreatorId())
                    .name(source.getCreatorName())
                    .build())
            .modifier(User.builder()
                    .id(source.getModifierId())
                    .name(source.getModifierName())
                    .build())
            .gmtCreate(source.getGmtCreate())
            .gmtModified(source.getGmtModified())
            .build();
    return target;
}
```

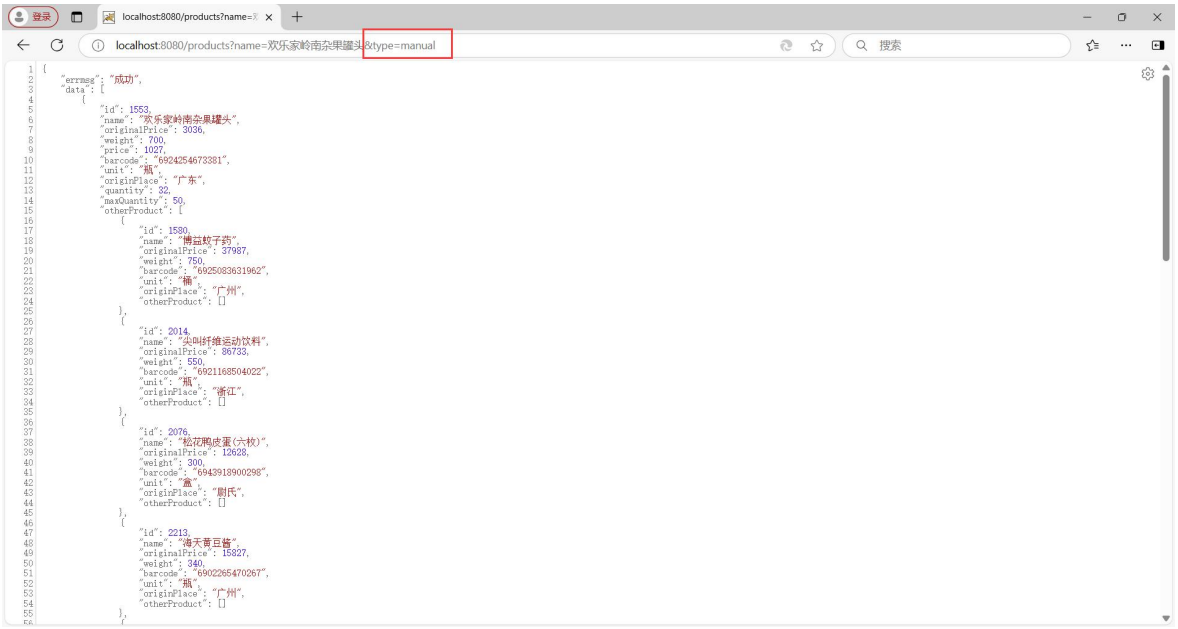
Ⅴ.最后在 dao 层新增 findProductByName_joinManual 方法，由于需要返回元素为 Product 类的 List 对象，我们需要先通过 mapper 层的查询方法查到 ProductJoinPo，然后通过 CloneFactory 转换成需要的对象，再进行封装操作，具体代码如下：

```java
/**
 * join 查询
 * @param name
 * @return
 * @throws BusinessException
 */
public List<Product> findProductByName_joinManual(String name) throws BusinessException {
    List<Product> productList = new ArrayList<>();
    List<ProductJoinPo> productJoinPoList = productJoinMapper.selectProductJoinByName(name);
    List<Product> otherProductList = new ArrayList<>();
    List<OnSale> onSaleList = new ArrayList<>();
    for (ProductJoinPo productJoinPo : productJoinPoList) {
        if (!productJoinPo.getName().equals(name)) { // 除去最开始通过 name 查到的那一行数据
            otherProductList.add(CloneFactory.copy(new Product(), productJoinPo));
        } else {
            productList.add(CloneFactory.copy(new Product(), productJoinPo));
            onSaleList.add(CloneFactory.copy(new OnSale(), productJoinPo));
        }
    }

    // 填充查询到的 product 的 otherProduct 属性和 onSale 属性
    productList.get(0).setOtherProduct(otherProductList);
    productList.get(0).setOnSaleList(onSaleList);
    return productList;
}
```

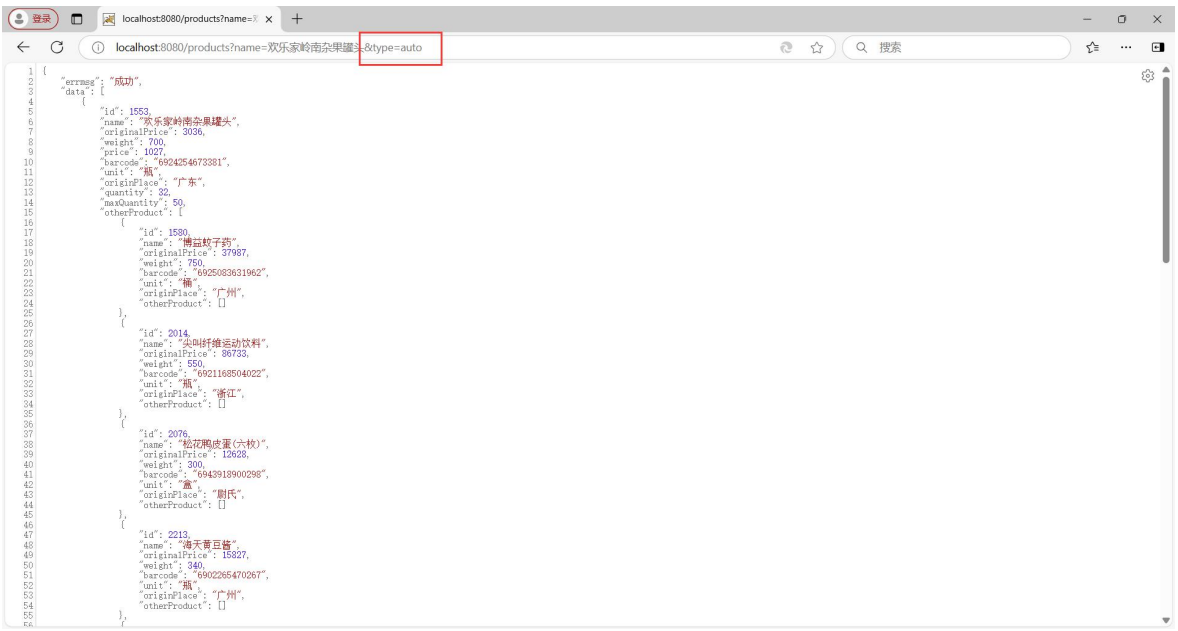
Ⅵ.本地测试结果：

manual 方式



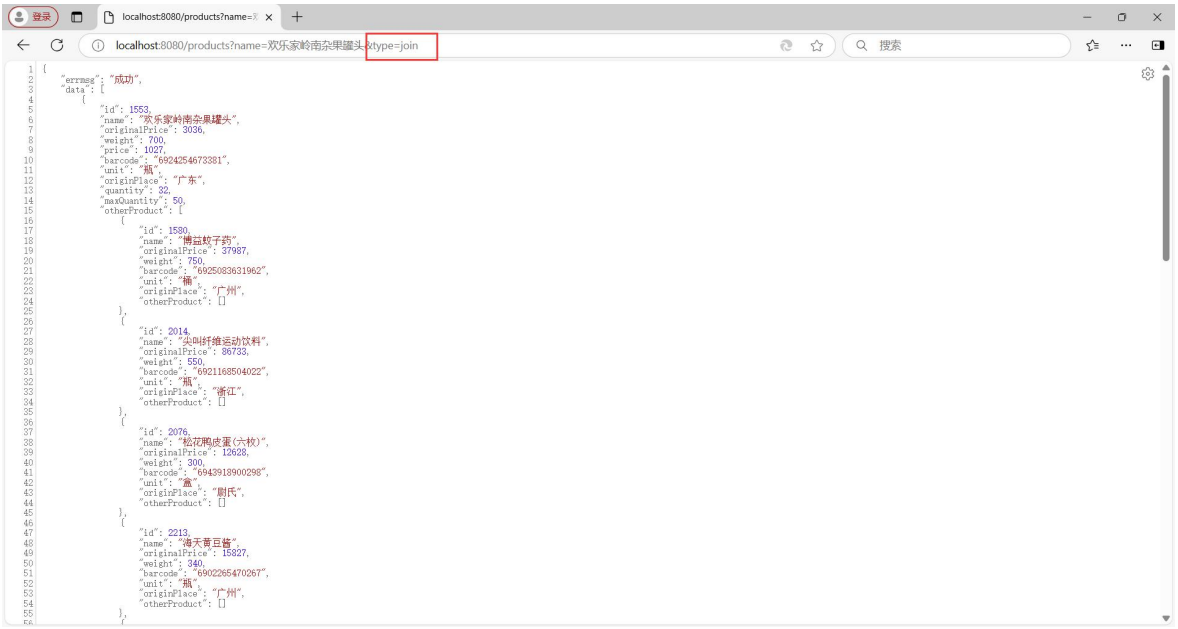

```
{
    "errmsg": "成功",
    "data": [
        {
            "id": 1553,
            "name": "欢乐家岭南杂果罐头",
            "originalPrice": 3036,
            "weight": 700,
            "price": 1027,
            "barcode": "6924254673381",
            "unit": "瓶",
            "originPlace": "广东",
            "quantity": 82,
            "maxQuantity": 50,
            "otherProduct": [
                {
                    "id": 1580,
                    "name": "博益蚊子药",
                    "originalPrice": 37987,
                    "weight": 750,
                    "barcode": "6925083631962",
                    "unit": "桶",
                    "originPlace": "广州",
                    "otherProduct": []
                },
                {
                    "id": 2014,
                    "name": "尖叫纤维运动饮料",
                    "originalPrice": 86733,
                    "weight": 550,
                    "barcode": "6921168504022",
                    "unit": "瓶",
                    "originPlace": "浙江",
                    "otherProduct": []
                },
                {
                    "id": 2076,
                    "name": "松花鸭皮蛋(六枚)",
                    "originalPrice": 12628,
                    "weight": 300,
                    "barcode": "6943918900298",
                    "unit": "盒",
                    "originPlace": "厨氏",
                    "otherProduct": []
                },
                {
                    "id": 2213,
                    "name": "海天黄豆酱",
                    "originalPrice": 15827,
                    "weight": 340,
                    "barcode": "6902265470267",
                    "unit": "瓶",
                    "originPlace": "广州",
                    "otherProduct": []
                },
```


auto 方式




```
{
    "errmsg": "成功",
    "data": [
        {
            "id": 1553,
            "name": "欢乐家岭南杂果罐头",
            "originalPrice": 3036,
            "weight": 700,
            "price": 1027,
            "barcode": "6924254673381",
            "unit": "瓶",
            "originPlace": "广东",
            "quantity": 82,
            "maxQuantity": 50,
            "otherProduct": [
                {
                    "id": 1580,
                    "name": "博益蚊子药",
                    "originalPrice": 37987,
                    "weight": 750,
                    "barcode": "6925083631962",
                    "unit": "桶",
                    "originPlace": "广州",
                    "otherProduct": []
                },
                {
                    "id": 2014,
                    "name": "尖叫纤维运动饮料",
                    "originalPrice": 86733,
                    "weight": 550,
                    "barcode": "6921168504022",
                    "unit": "瓶",
                    "originPlace": "浙江",
                    "otherProduct": []
                },
                {
                    "id": 2076,
                    "name": "松花鸭皮蛋(六枚)",
                    "originalPrice": 12628,
                    "weight": 300,
                    "barcode": "6943918900298",
                    "unit": "盒",
                    "originPlace": "厨氏",
                    "otherProduct": []
                },
                {
                    "id": 2213,
                    "name": "海天黄豆酱",
                    "originalPrice": 15827,
                    "weight": 340,
                    "barcode": "6902265470267",
                    "unit": "瓶",
                    "originPlace": "广州",
                    "otherProduct": []
                },
```

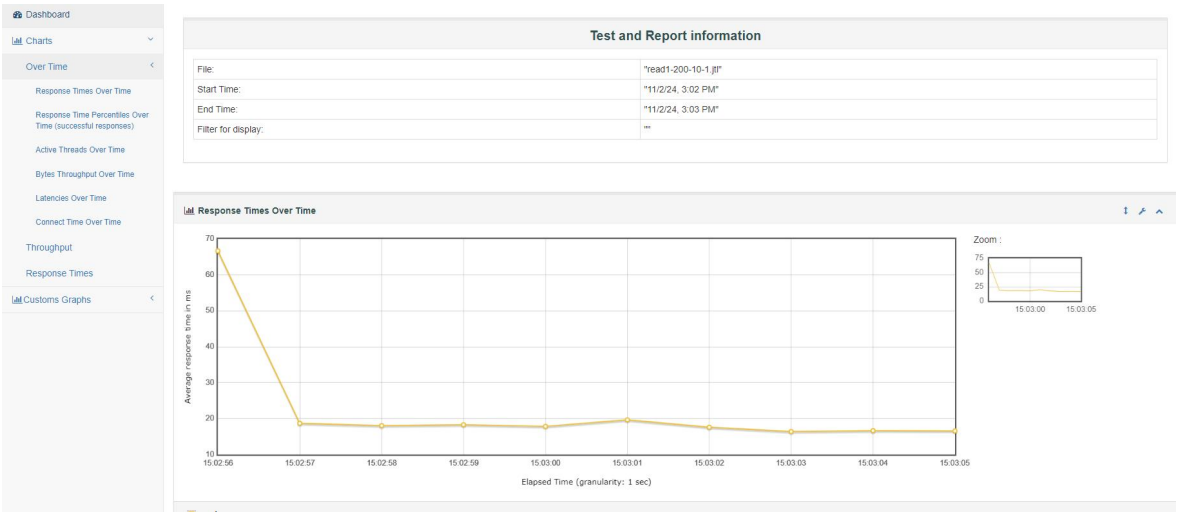

join 方式

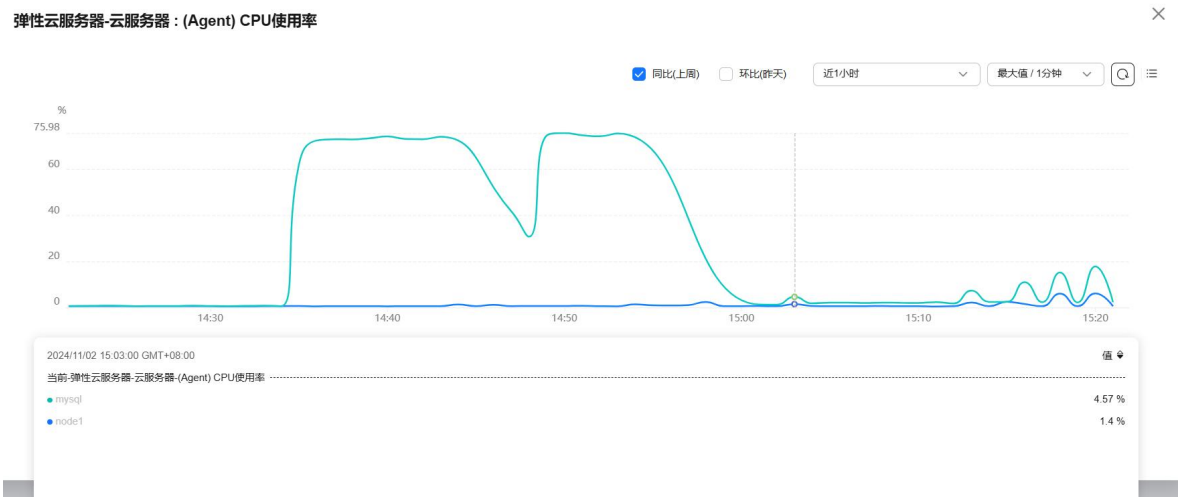


(3) 通过 Jmeter 测试，对比三种访问方式查询效率不同

Auto：

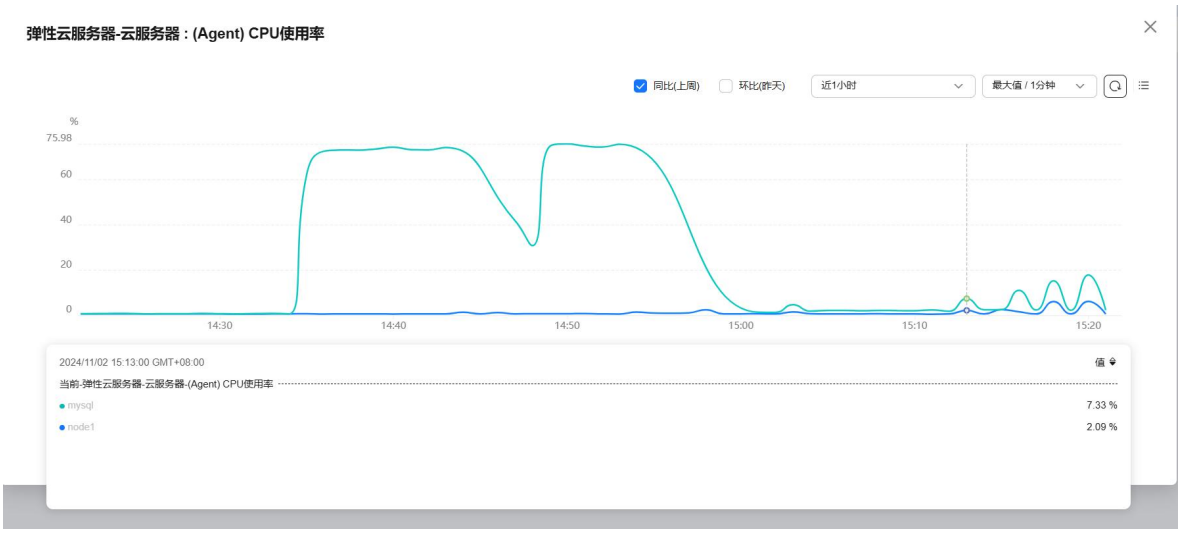
Thread=200



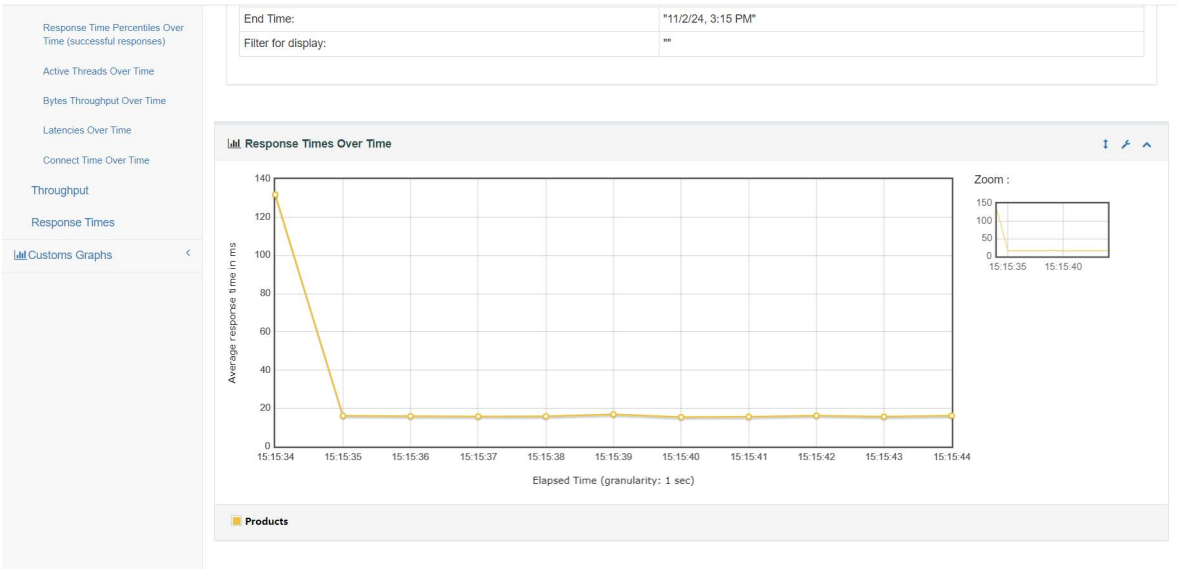

弹性云服务器-云服务器 : (Agent) CPU使用率
同比(上周)
环比(昨天)
近1小时
最大值 / 1分钟
%
75.98
60
40
20
0
14:30
14:40
14:50
15:00
15:10
15:20
2024/11/02 15:03:00 GMT+08:00
值
当前-弹性云服务器-云服务器-(Agent) CPU使用率
mysql
4.57 %
node1
1.4 %

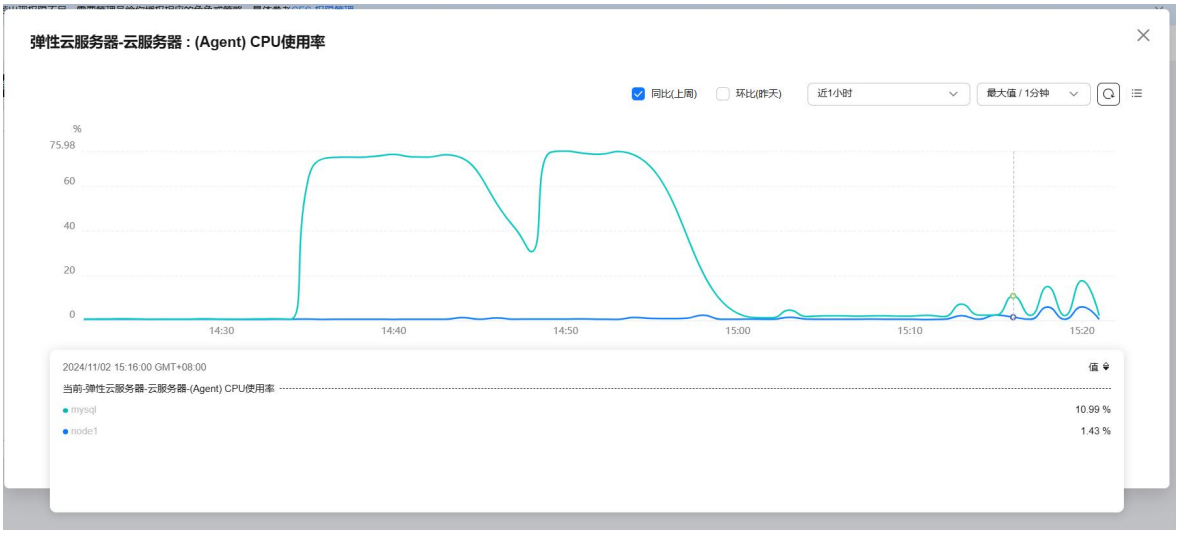
Thread=400


Response Time Percentiles Over Time (successful responses)
Active Threads Over Time
Bytes Throughput Over Time
Latencies Over Time
Connect Time Over Time
Throughput
Response Times
Customs Graphs
End Time:
"11/2/24, 3:13 PM"
Filter for display:
""
Response Times Over Time
Zoom :
150
100
50
0
15:13:00
15:13:05
15:13:10
Average response time in ms
120
100
80
60
40
20
0
15:13:00
15:13:01
15:13:02
15:13:03
15:13:04
15:13:05
15:13:06
15:13:07
15:13:08
15:13:09
15:13:10
Elapsed Time (granularity: 1 sec)
Products

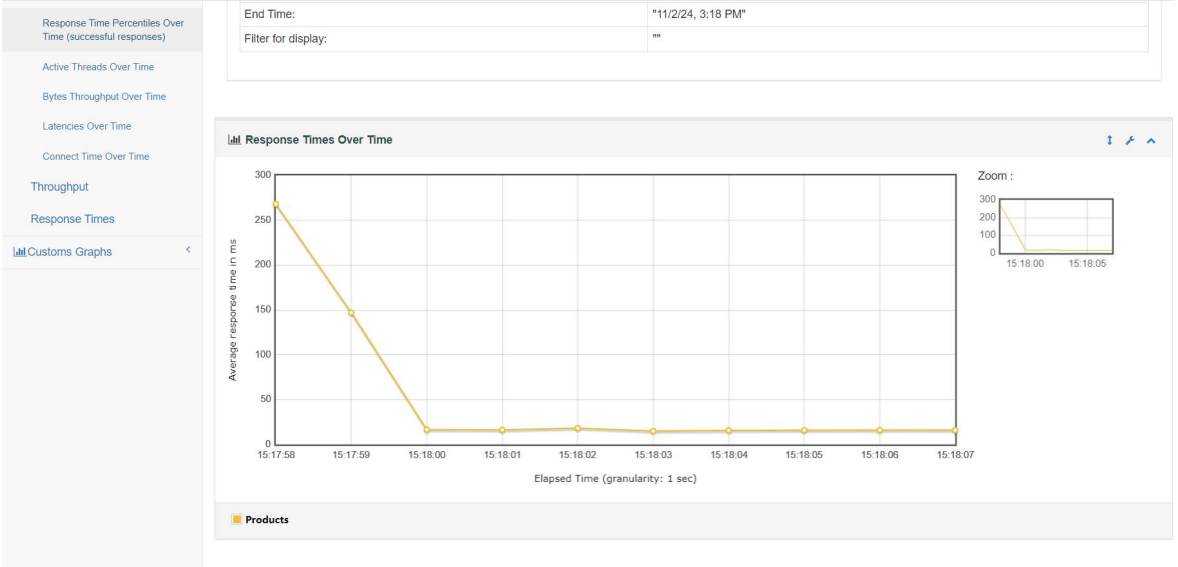

弹性云服务器-云服务器 : (Agent) CPU使用率
同比(上周)
环比(昨天)
近1小时
最大值 / 1分钟
%
75.98
60
40
20
0
14:30
14:40
14:50
15:00
15:10
15:20
2024/11/02 15:13:00 GMT+08:00
值
当前-弹性云服务器-云服务器-(Agent) CPU使用率
mysql
7.33 %
node1
2.09 %

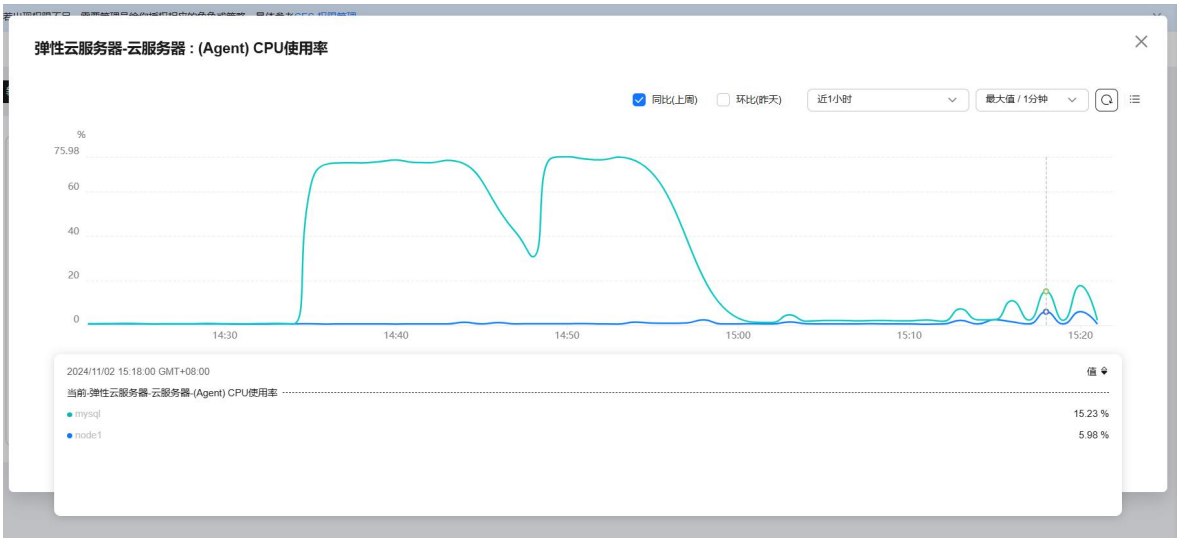
Thread=800


Response Time Percentiles Over Time (successful responses)
Active Threads Over Time
Bytes Throughput Over Time
Latencies Over Time
Connect Time Over Time
Throughput
Response Times
Customs Graphs
End Time:
"11/2/24, 3:15 PM"
Filter for display:
""
Response Times Over Time
Zoom :
150
100
50
0
15:15:35
15:15:40
Average response time in ms
140
120
100
80
60
40
20
0
15:15:34
15:15:35
15:15:36
15:15:37
15:15:38
15:15:39
15:15:40
15:15:41
15:15:42
15:15:43
15:15:44
Elapsed Time (granularity: 1 sec)
Products

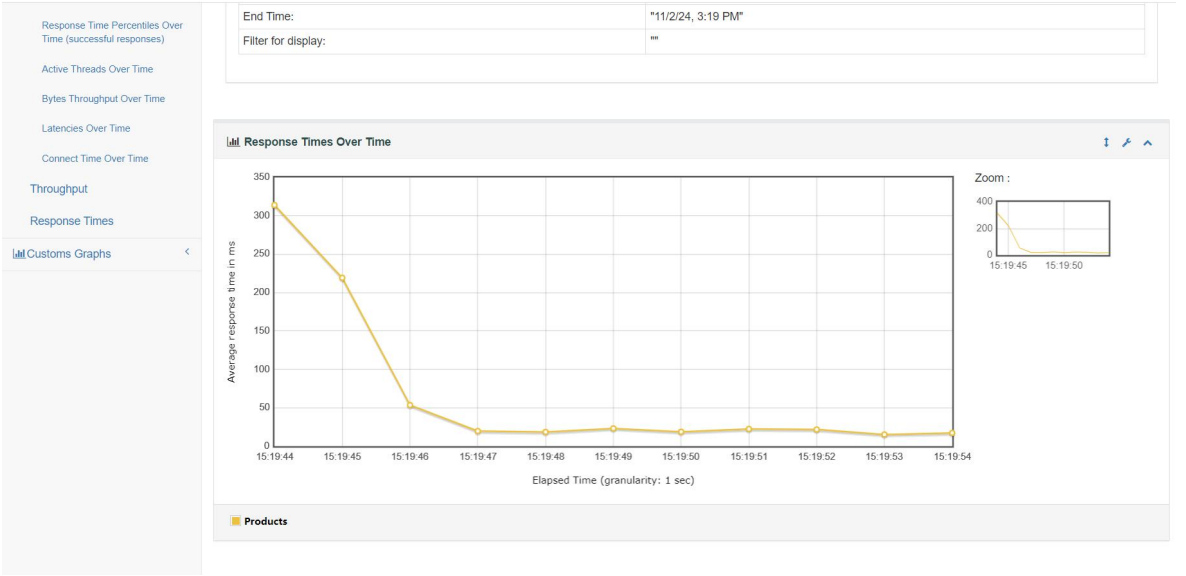

弹性云服务器-云服务器 : (Agent) CPU使用率
同比(上周)
环比(昨天)
近1小时
最大值 / 1分钟
%
75.98
60
40
20
0
14:30
14:40
14:50
15:00
15:10
15:20
2024/11/02 15:16:00 GMT+08:00
值
当前-弹性云服务器-云服务器-(Agent) CPU使用率
mysql
10.99 %
node1
1.43 %

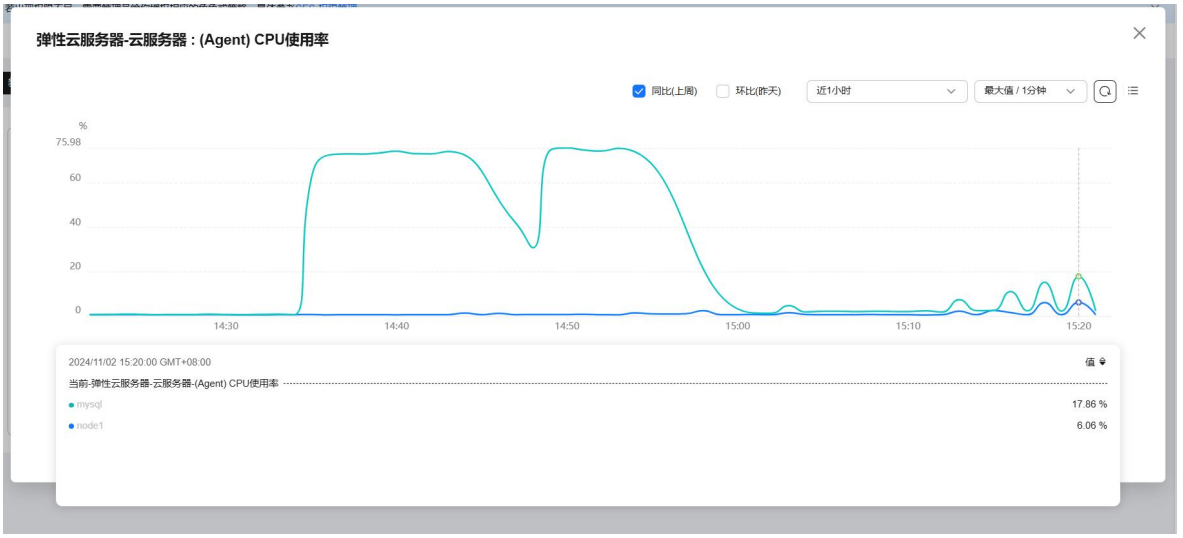
Thread=1200


Response Time Percentiles Over Time (successful responses)
Active Threads Over Time
Bytes Throughput Over Time
Latencies Over Time
Connect Time Over Time
Throughput
Response Times
Customs Graphs
End Time:
"11/2/24, 3:18 PM"
Filter for display:
""
Response Times Over Time
Zoom :
Average response time in ms
Elapsed Time (granularity: 1 sec)
15:17:58
15:17:59
15:18:00
15:18:01
15:18:02
15:18:03
15:18:04
15:18:05
15:18:06
15:18:07
Products


弹性云服务器-云服务器 : (Agent) CPU使用率
同比(上周)
环比(昨天)
近1小时
最大值 / 1分钟
%
75.98
60
40
20
0
14:30
14:40
14:50
15:00
15:10
15:20
2024/11/02 15:18:00 GMT+08:00
当前-弹性云服务器-云服务器-(Agent) CPU使用率
mysql
15.23 %
node1
5.98 %

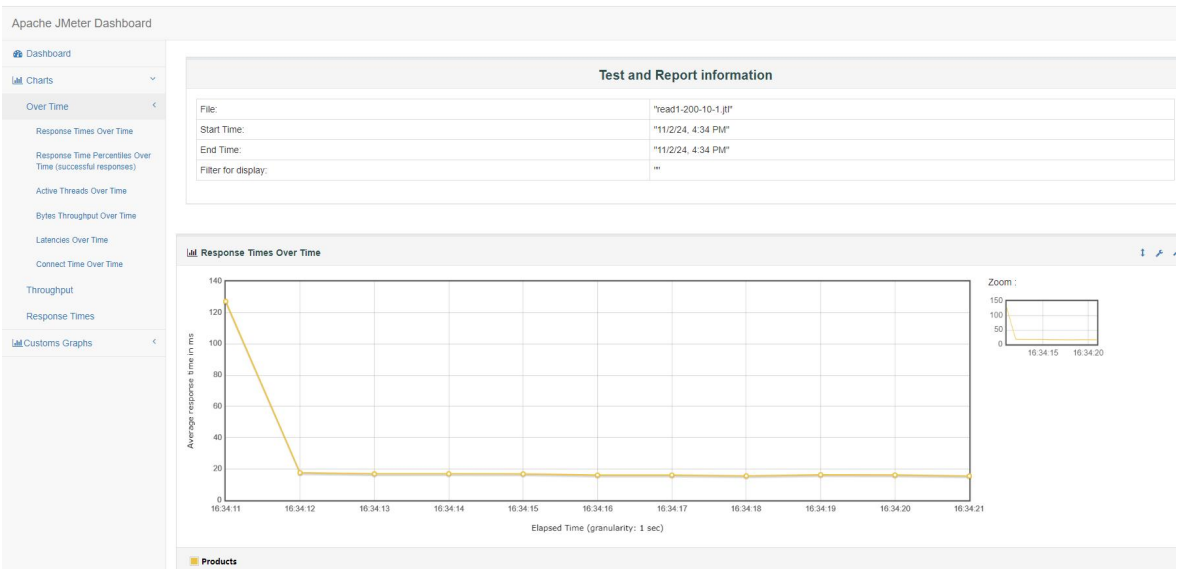
Thread=1500

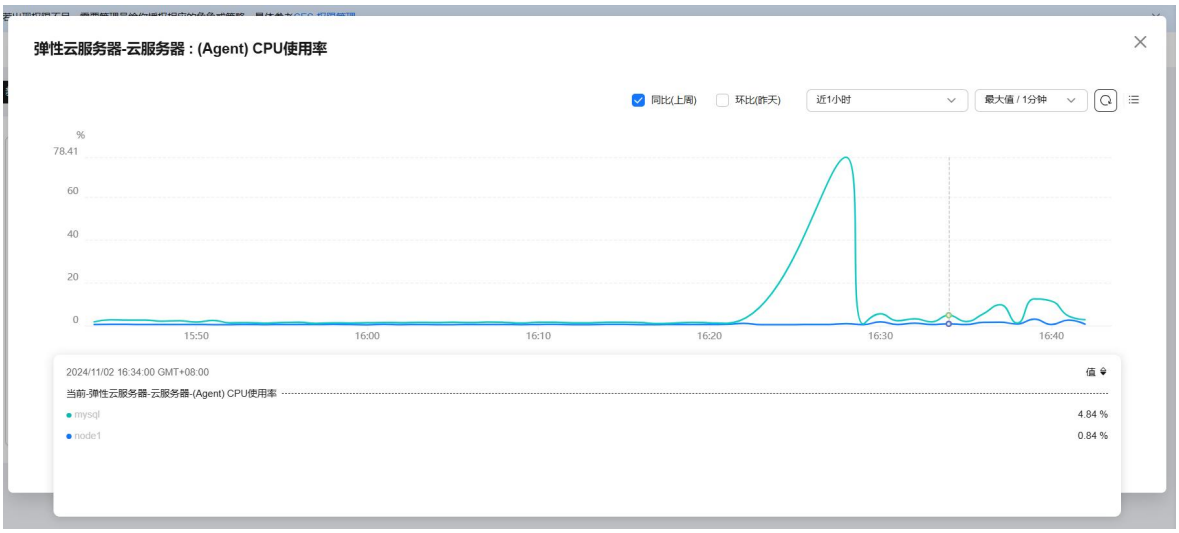

Response Time Percentiles Over Time (successful responses)
Active Threads Over Time
Bytes Throughput Over Time
Latencies Over Time
Connect Time Over Time
Throughput
Response Times
Customs Graphs
End Time:
"11/2/24, 3:19 PM"
Filter for display:
""
Response Times Over Time
Zoom :
400
200
0
15:19:45
15:19:50
Average response time in ms
350
300
250
200
150
100
50
0
15:19:44
15:19:45
15:19:46
15:19:47
15:19:48
15:19:49
15:19:50
15:19:51
15:19:52
15:19:53
15:19:54
Elapsed Time (granularity: 1 sec)
Products



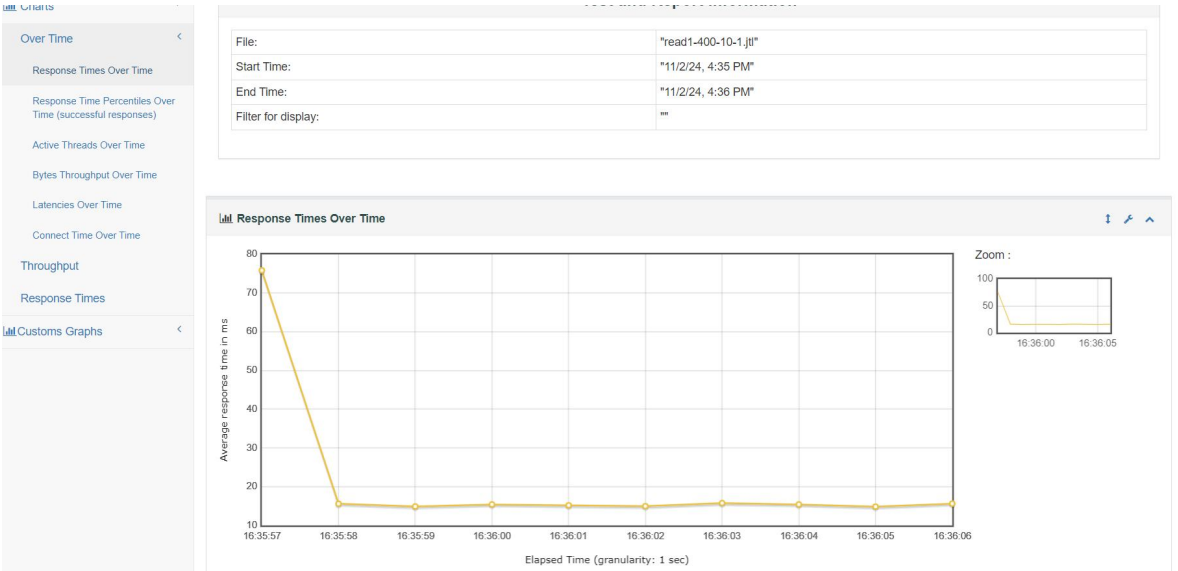
Manual：

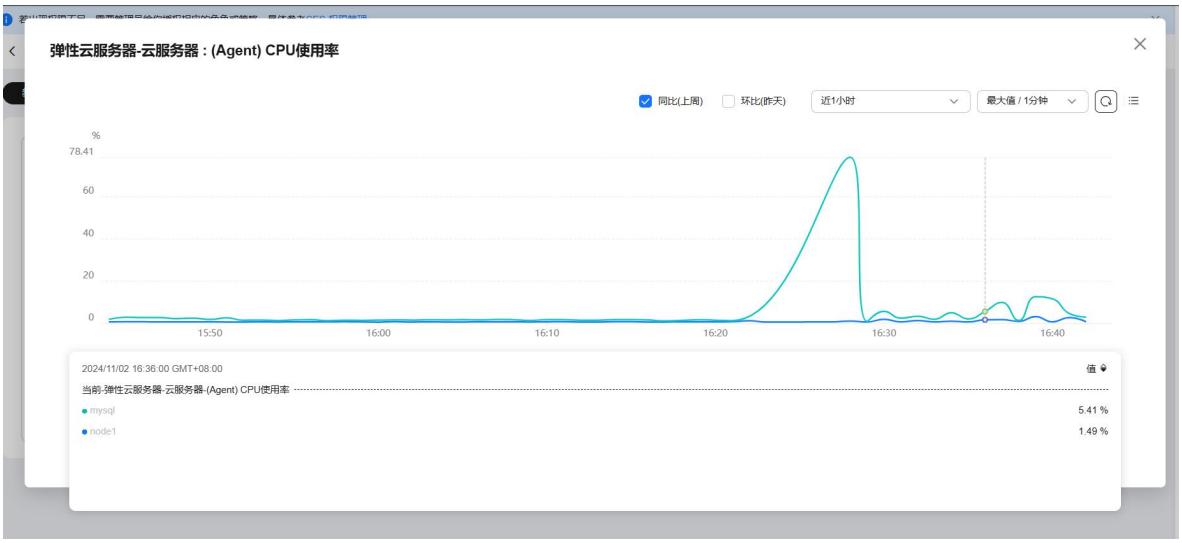
Thread=200




弹性云服务器-云服务器：(Agent) CPU使用率
同比(上周)
环比(昨天)
近1小时
最大值 / 1分钟
%
78.41
60
40
20
0
15:50
16:00
16:10
16:20
16:30
16:40
2024/11/02 16:34:00 GMT+08:00
值
当前-弹性云服务器-云服务器-(Agent) CPU使用率
mysql
4.84 %
node1
0.84 %

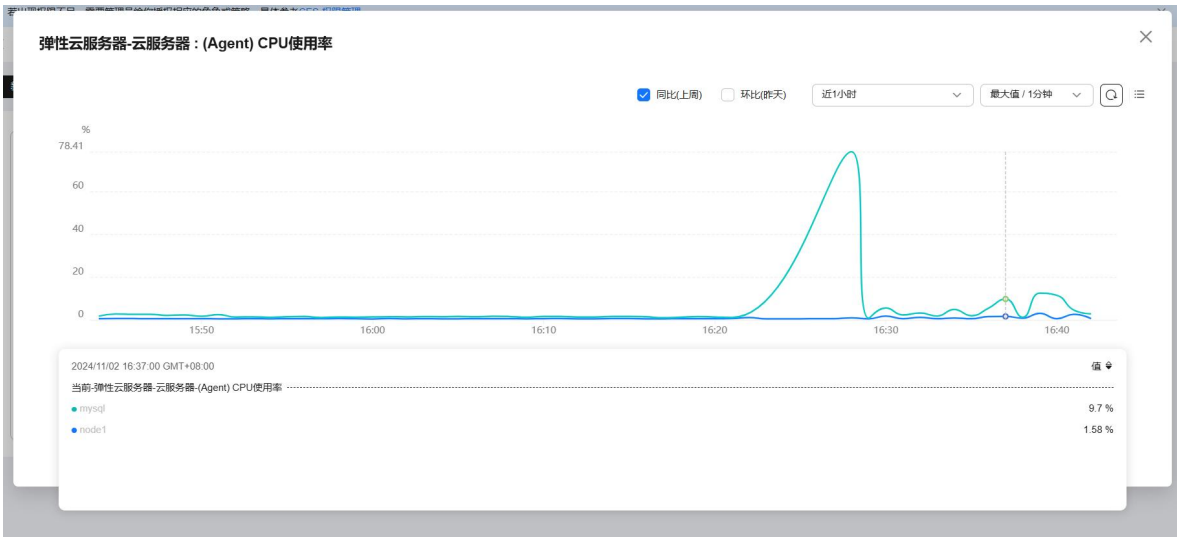
Thread=400


Charts
Over Time
Response Times Over Time
Response Time Percentiles Over Time (successful responses)
Active Threads Over Time
Bytes Throughput Over Time
Latencies Over Time
Connect Time Over Time
Throughput
Response Times
Customs Graphs
File: "read1-400-10-1.jtl"
Start Time: "11/2/24, 4:35 PM"
End Time: "11/2/24, 4:36 PM"
Filter for display: ""
Response Times Over Time
Zoom :
Average response time in ms
Elapsed Time (granularity: 1 sec)


弹性云服务器-云服务器 : (Agent) CPU使用率
同比(上周)
环比(昨天)
近1小时
最大值 / 1分钟
%
78.41
60
40
20
0
15:50
16:00
16:10
16:20
16:30
16:40
2024/11/02 16:36:00 GMT+08:00
值
当前-弹性云服务器-云服务器-(Agent) CPU使用率
mysql
5.41 %
node1
1.49 %

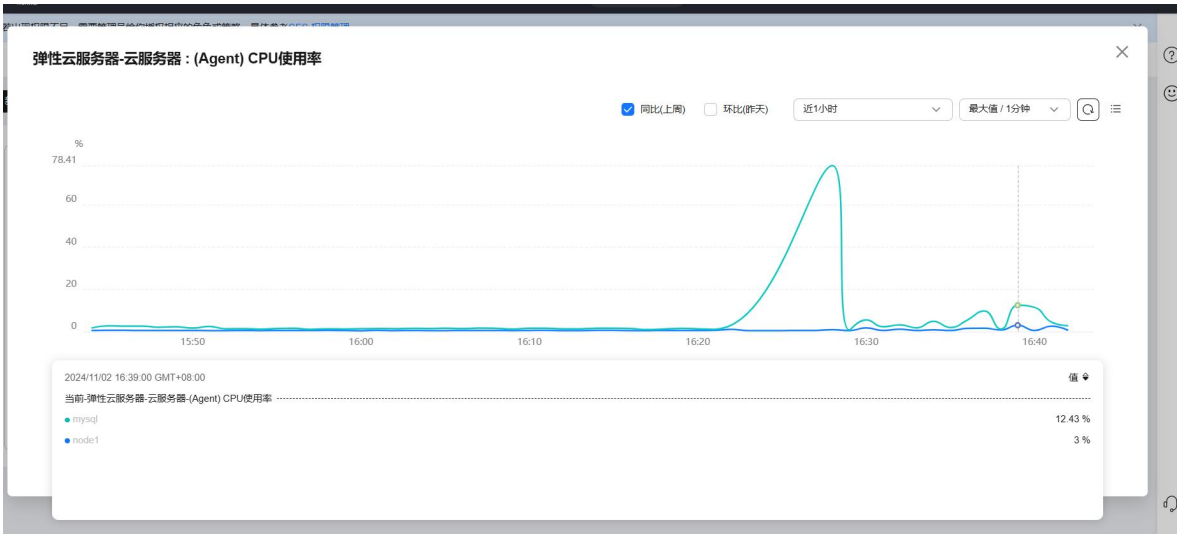
Thread=800


Response Time Percentiles Over Time (successful responses)
Active Threads Over Time
Bytes Throughput Over Time
Latencies Over Time
Connect Time Over Time
Throughput
Response Times
Customs Graphs
End Time:
"11/2/24, 4:37 PM"
Filter for display:
Response Times Over Time
Zoom :
150
100
50
0
16:37:35
16:37:40
120
100
80
60
40
20
0
Average response time in ms
16:37:33
16:37:34
16:37:35
16:37:36
16:37:37
16:37:38
16:37:39
16:37:40
16:37:41
16:37:42
16:37:43
Elapsed Time (granularity: 1 sec)
Products

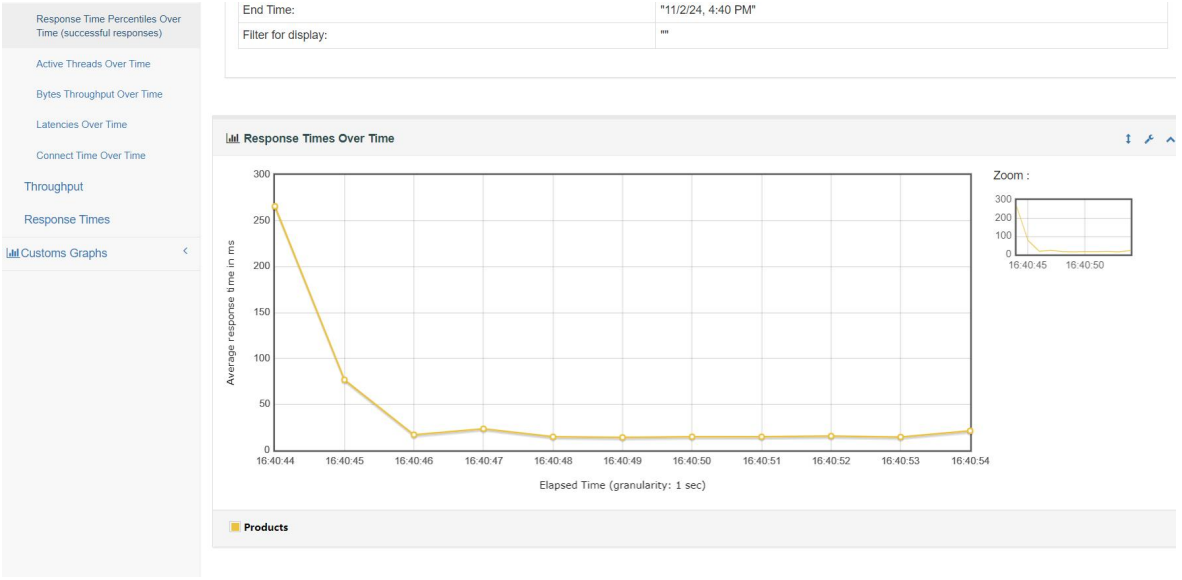

弹性云服务器-云服务器 : (Agent) CPU使用率
同比(上周)
环比(昨天)
近1小时
最大值 / 1分钟
%
78.41
60
40
20
0
15:50
16:00
16:10
16:20
16:30
16:40
2024/11/02 16:37:00 GMT+08:00
值
当前-弹性云服务器-云服务器-(Agent) CPU使用率
mysql
9.7 %
node1
1.58 %

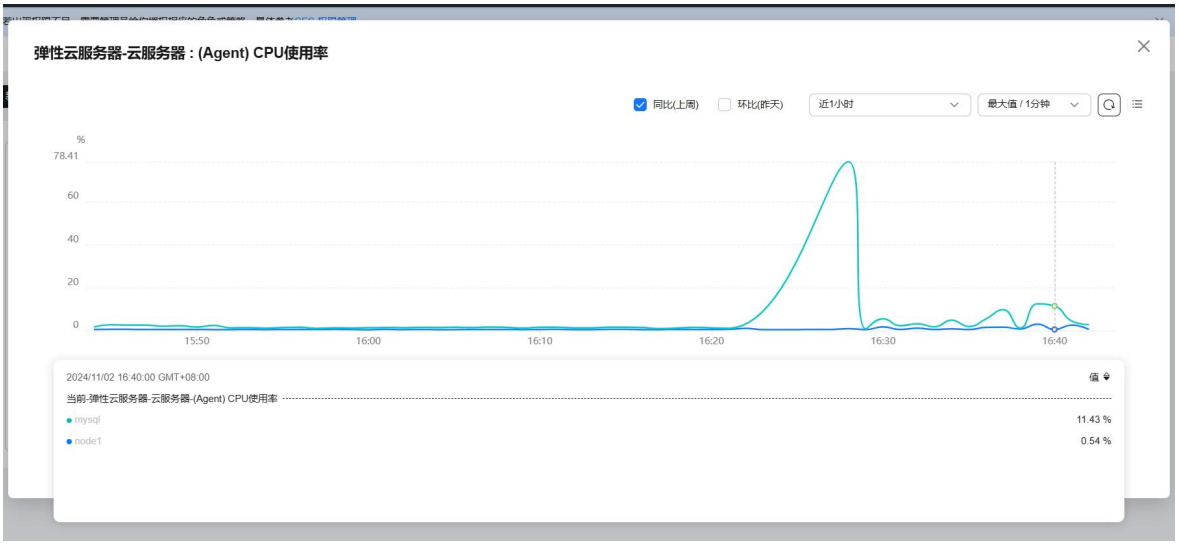
Thread=1200


Response Time Percentiles Over Time (successful responses)
Active Threads Over Time
Bytes Throughput Over Time
Latencies Over Time
Connect Time Over Time
Throughput
Response Times
Customs Graphs
End Time:
Filter for display:
Response Times Over Time
Zoom :
Average response time in ms
Elapsed Time (granularity: 1 sec)
Products


弹性云服务器-云服务器 : (Agent) CPU使用率
同比(上周)
环比(昨天)
近1小时
最大值 / 1分钟
%
78.41
60
40
20
0
15:50
16:00
16:10
16:20
16:30
16:40
2024/11/02 16:39:00 GMT+08:00
值
当前-弹性云服务器-云服务器-(Agent) CPU使用率
mysql
12.43 %
node1
3 %

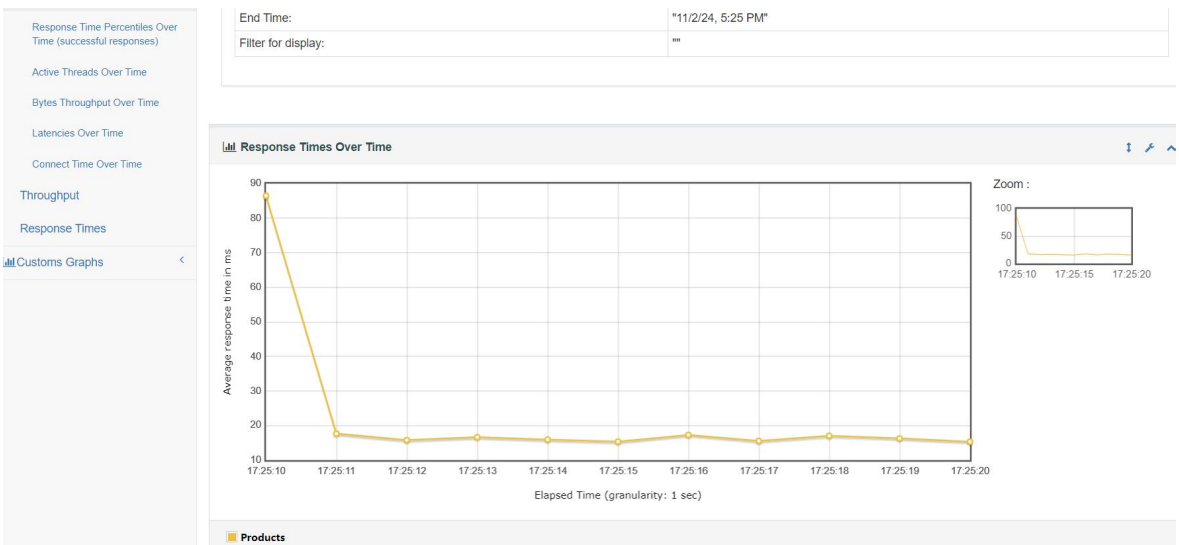
Thread=1500

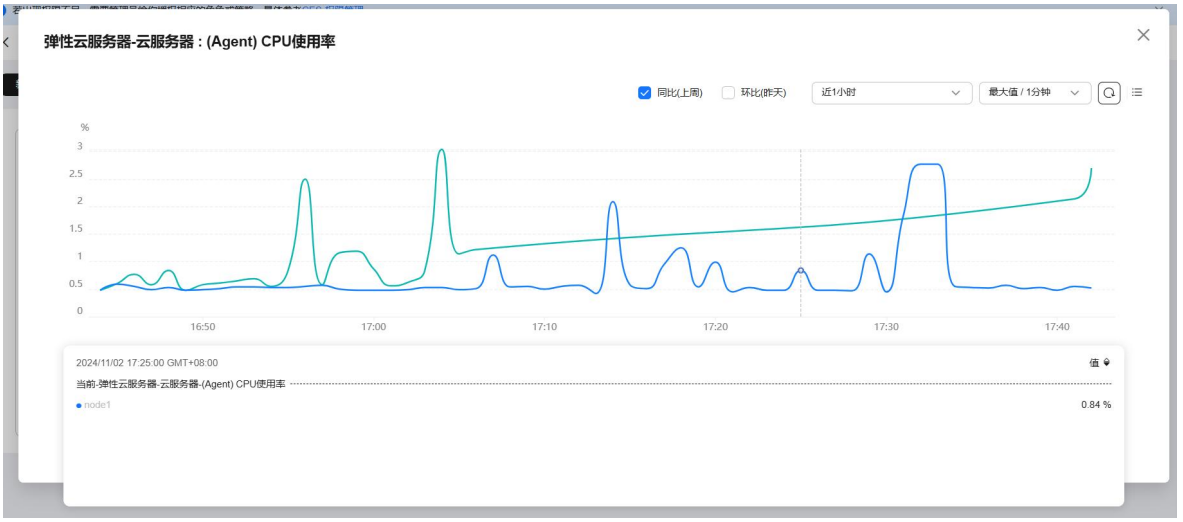

Response Time Percentiles Over Time (successful responses)
Active Threads Over Time
Bytes Throughput Over Time
Latencies Over Time
Connect Time Over Time
Throughput
Response Times
Customs Graphs
End Time:
"11/2/24, 4:40 PM"
Filter for display:
""
Response Times Over Time
Zoom :
300
200
100
0
16:40:45
16:40:50
Average response time in ms
300
250
200
150
100
50
0
16:40:44
16:40:45
16:40:46
16:40:47
16:40:48
16:40:49
16:40:50
16:40:51
16:40:52
16:40:53
16:40:54
Elapsed Time (granularity: 1 sec)
Products



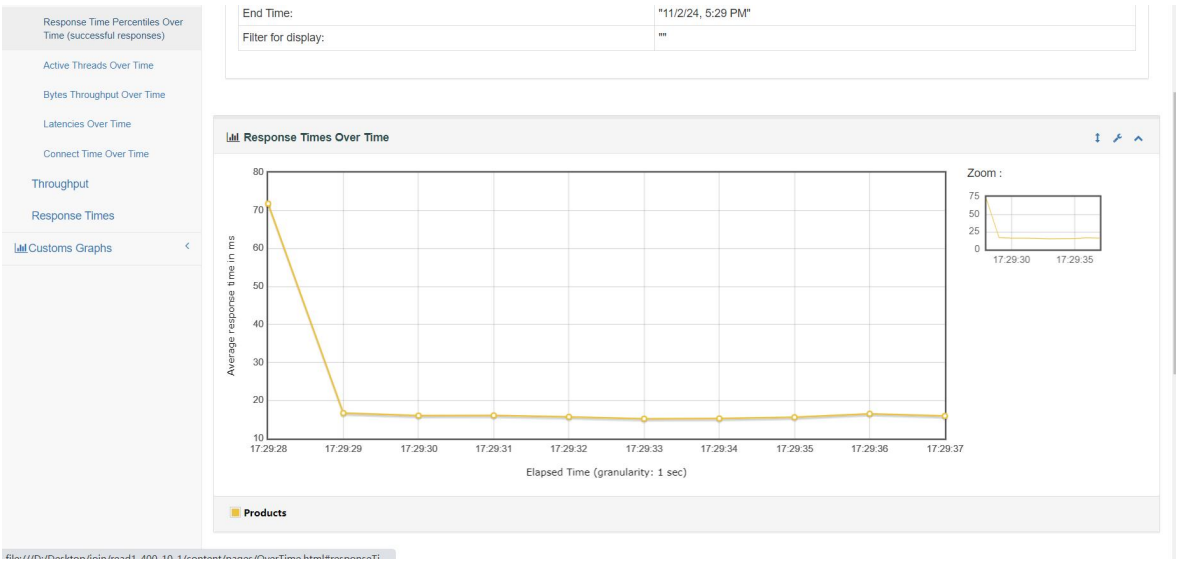
Join：

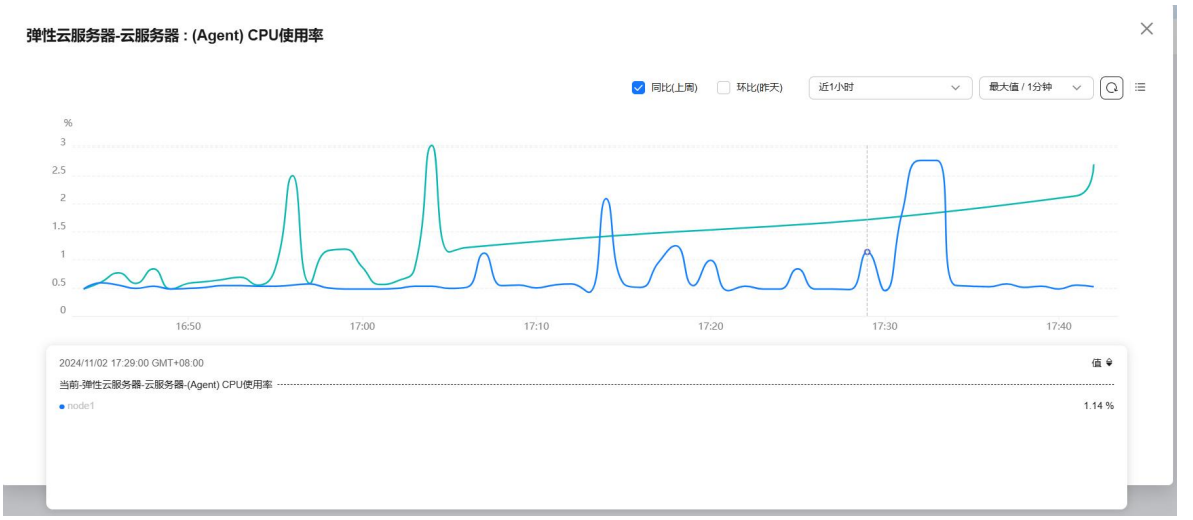
Thread=200



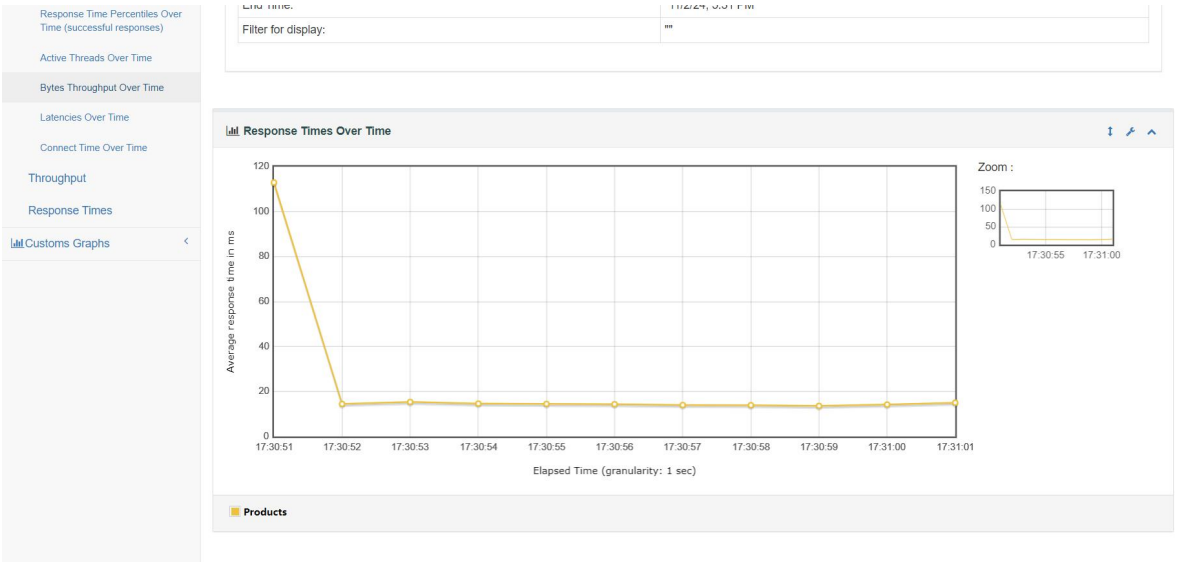

弹性云服务器-云服务器 : (Agent) CPU使用率
同比(上周)
环比(昨天)
近1小时
最大值 / 1分钟
%
3
2.5
2
1.5
1
0.5
0
16:50
17:00
17:10
17:20
17:30
17:40
2024/11/02 17:25:00 GMT+08:00
当前-弹性云服务器-云服务器-(Agent) CPU使用率
node1
0.84 %

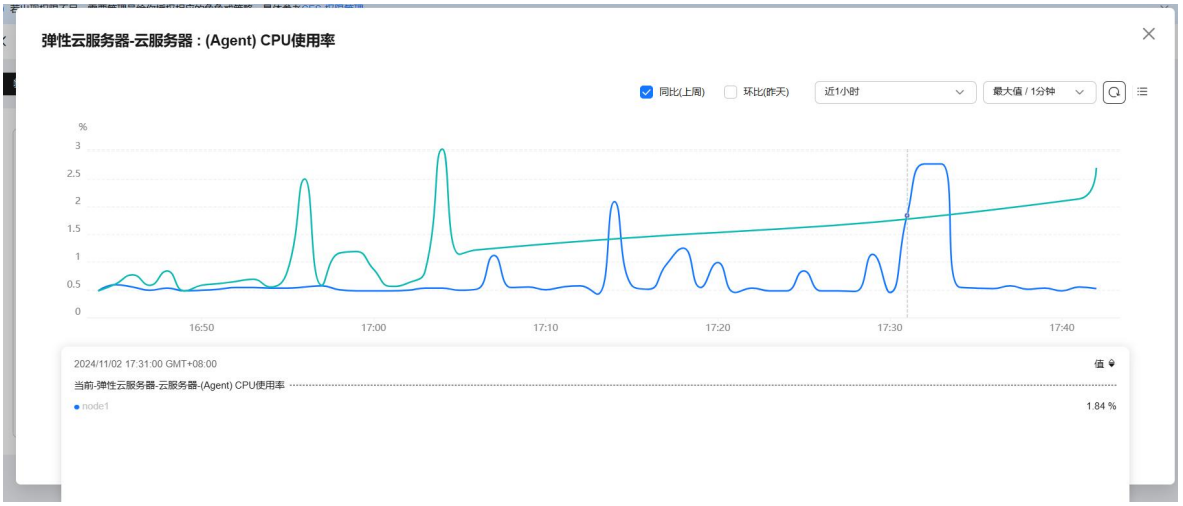
Thread=400


Response Time Percentiles Over Time (successful responses)
Active Threads Over Time
Bytes Throughput Over Time
Latencies Over Time
Connect Time Over Time
Throughput
Response Times
Customs Graphs
End Time:
"11/2/24, 5:29 PM"
Filter for display:
""
Response Times Over Time
Zoom :
Average response time in ms
Elapsed Time (granularity: 1 sec)
Products

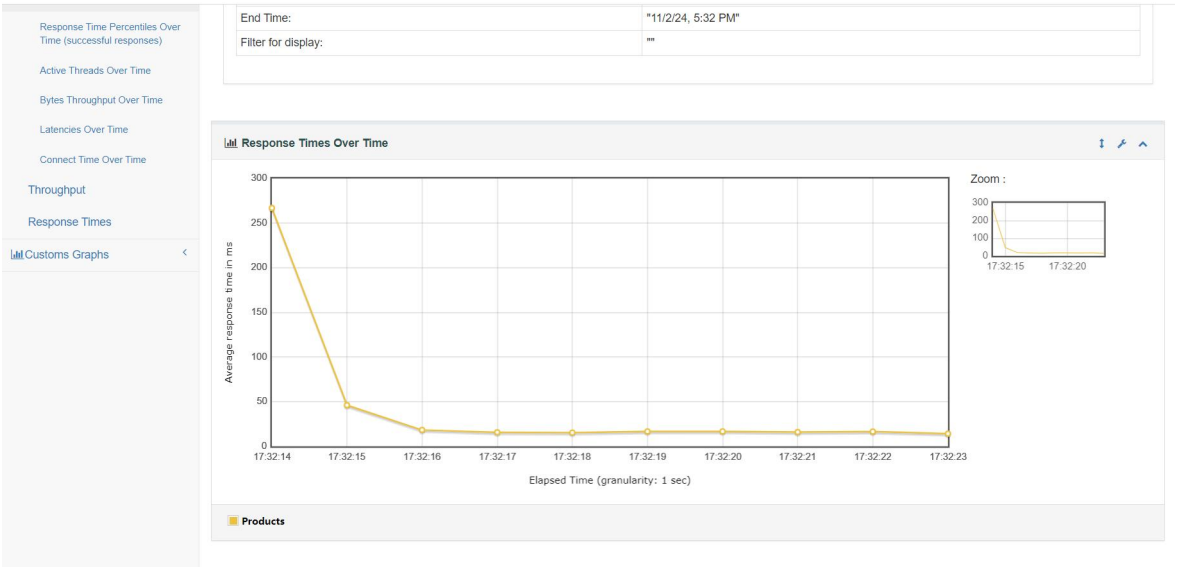

弹性云服务器-云服务器 : (Agent) CPU使用率
同比(上周)
环比(昨天)
近1小时
最大值 / 1分钟
%
3
2.5
2
1.5
1
0.5
0
16:50
17:00
17:10
17:20
17:30
17:40
2024/11/02 17:29:00 GMT+08:00
值
当前-弹性云服务器-云服务器-(Agent) CPU使用率
node1
1.14 %

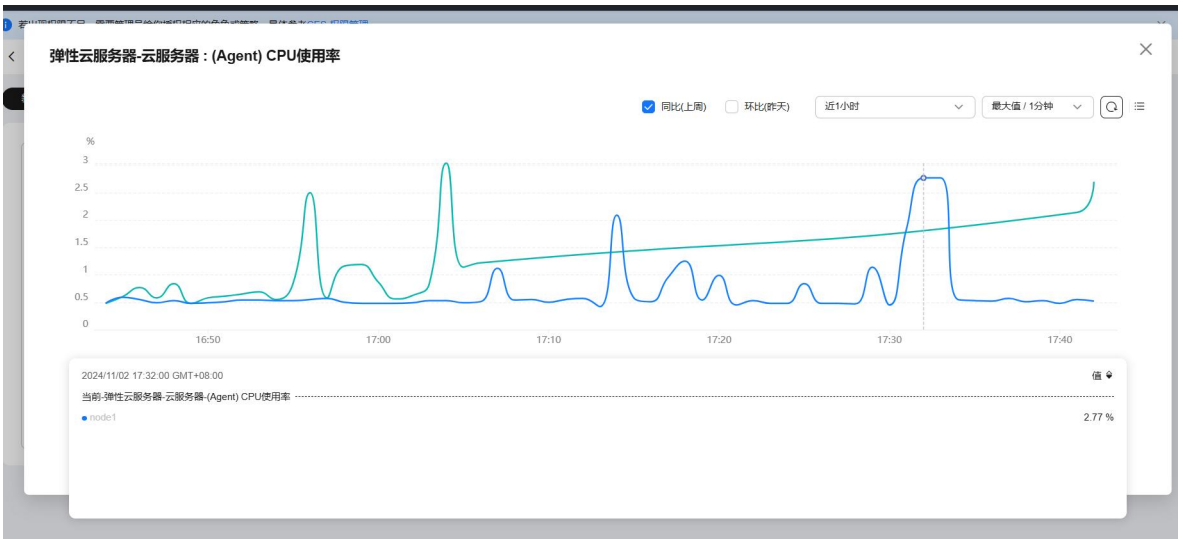
Thread=800


Response Time Percentiles Over Time (successful responses)
Active Threads Over Time
Bytes Throughput Over Time
Latencies Over Time
Connect Time Over Time
Throughput
Response Times
Customs Graphs
Filter for display:
""
Response Times Over Time
Zoom :
Average response time in ms
Elapsed Time (granularity: 1 sec)
Products

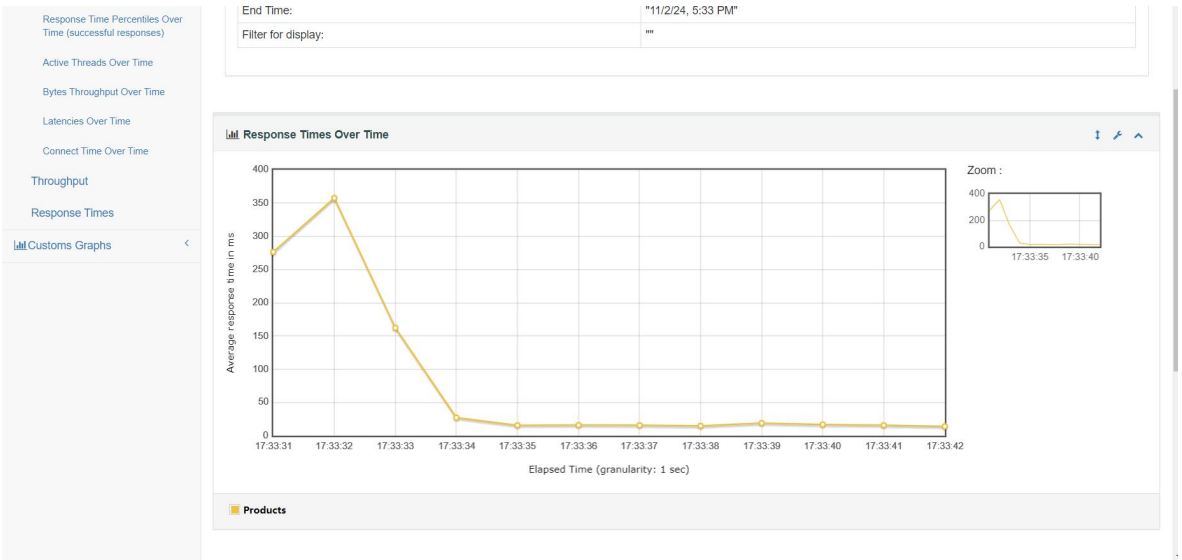

弹性云服务器-云服务器 : (Agent) CPU使用率
同比(上周)
环比(昨天)
近1小时
最大值 / 1分钟
%
3
2.5
2
1.5
1
0.5
0
16:50
17:00
17:10
17:20
17:30
17:40
2024/11/02 17:31:00 GMT+08:00
值
当前-弹性云服务器-云服务器-(Agent) CPU使用率
node1
1.84 %

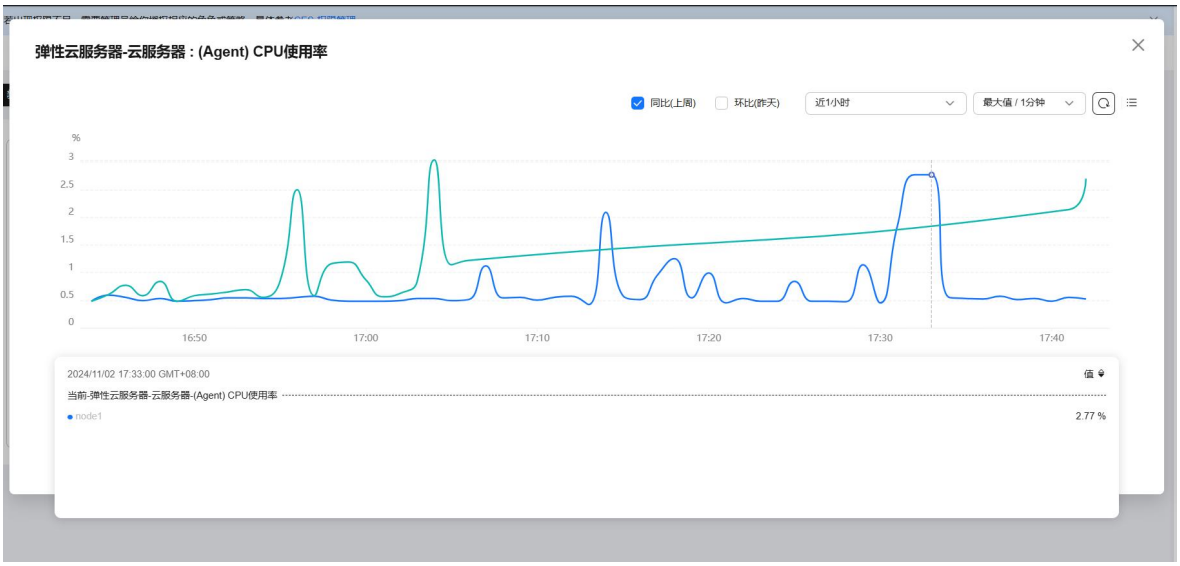
Thread=1200


Response Time Percentiles Over Time (successful responses)
Active Threads Over Time
Bytes Throughput Over Time
Latencies Over Time
Connect Time Over Time
Throughput
Response Times
Customs Graphs
End Time:
"11/2/24, 5:32 PM"
Filter for display:
""
Response Times Over Time
Zoom :
Average response time in ms
Elapsed Time (granularity: 1 sec)
Products


弹性云服务器-云服务器 : (Agent) CPU使用率
同比(上周)
环比(昨天)
近1小时
最大值 / 1分钟
%
3
2.5
2
1.5
1
0.5
0
16:50
17:00
17:10
17:20
17:30
17:40
2024/11/02 17:32:00 GMT+08:00
值
当前-弹性云服务器-云服务器-(Agent) CPU使用率
node1
2.77 %

Thread=1500


Response Time Percentiles Over Time (successful responses)
Active Threads Over Time
Bytes Throughput Over Time
Latencies Over Time
Connect Time Over Time
Throughput
Response Times
Customs Graphs
End Time:
"11/2/24, 5:33 PM"
Filter for display:
""
Response Times Over Time
Zoom :
400
200
0
17:33:35
17:33:40
Average response time in ms
400
350
300
250
200
150
100
50
0
17:33:31
17:33:32
17:33:33
17:33:34
17:33:35
17:33:36
17:33:37
17:33:38
17:33:39
17:33:40
17:33:41
17:33:42
Elapsed Time (granularity: 1 sec)
Products



结果分析与讨论：

auto 随着 Thread 变化，使用率相差较小。
Manual 在 thread 较大时，变化率也变大，而在 1200 至 1500 时，使用率下降。
Join 使用率较低，变化率也更大。

auto 的响应时间为：thread=200—65+ms
thread=400—110+ms
thread=800—130+ms
thread=1200—250+ms
thread=1500—300+ms
且在 thread=800 及以上时，auto 响应需要持续 2s。

manual 的响应时间为：thread=200—65+ms
thread=400—75+ms
thread=800—100+ms
thread=1200—200+ms
thread=1500—250+ms

join 的响应时间为：thread=200—60+ms
thread=400—70+ms
thread=800—100+ms
thread=1200—200+ms
thread=1500—250+ms

与 auto 相比，join 的响应时间较短。在 thread=800 以下，响应时间一次提高大致 10ms，

在 thread=1200 以上，响应时间一次提高大致 50ms。

与 manual 相比，join 在 thread=400 以下时，响应时间一次提高大致 5ms，在 thread=800 以上时，响应时间差不多。而 join 在 thread=1200 时响应持续较短，join 在 thread 高时更稳定。

总结

通过对三种不同方式实现数据表关联的性能测试，我们发现使用 JOIN 查询的方式在处理大量并发请求时表现出更好的性能和更低的延迟。此外，JOIN 查询减少了对数据库的访问次数，从而降低了服务器的负载。虽然 JOIN 查询可能使得 SQL 语句更加复杂，但在高并发场景下，它能提供更优的服务质量。因此，在设计需要频繁进行数据表关联的应用时，推荐优先考虑使用 JOIN 查询的方式。

References:

[1] MyBatis 官方文档. Retrieved from https://mybatis.org/mybatis-3/index.html

[2] Spring Boot 官方文档. Retrieved from https://spring.io/projects/spring-boot

[3] JMeter 官方文档. Retrieved from https://jmeter.apache.org/usermanual/

[4] 华为云云监控服务文档. Retrieved from https://support.huaweicloud.com/userguides-cloudmonitoring/cm-overview.html