

**DolphinDB与MongoDB**

**基准测试报告**

**浙江智臾科技有限公司**

**2019.02**

**1.概述**

DolphinDB是一个高性能分布式时序数据库,属于列式内存关系型数据库，由c++语言编写。DolphinDB具有内置的并行和分布式计算性能，可用于实时数据处理和多节点计算分析。DolphinDB以列为存储单元，采用列式混合引擎存储结构化数据，可以处理海量的数据。

MongoDB是一个文档型的NoSQL数据库，由c++语言编写。MongoDB以文档为存储单元，按数据库，集合，文档的树型结构进行存储。它支持的数据结构较为松散，是类似JSON的BSON格式和DolphinDB中的列式结构化数据有着较大的差别。

MongoDB作为一种时下很流行的文档型数据库，其定位介于[关系数据库](https://baike.baidu.com/item/关系数据库)和非关系数据库之间，可以处理非结构化的数据和结构化的数据。MongoDB平时多用于处理互联网信息和网站文件等非结构化数据的情况，那对于银行或者会计系统等金融时序数据的处理，其性能表现又如何呢？为了展示MongoDB在处理结构化时序数据时的性能，我们使用大小两种数量级的结构化数据，对两种数据库进行综合的性能对比测试，测试内容包括：数据库单表查询，连接查询，计算，数据导入和导出，磁盘空间占用大小。

测试结果显示：在处理结构化数据时，DolphinDB比MongoDB查询速度要快，所占空间要小，综合性能要优越。具体优势体现在如下几个方面：

1、查询性能方面，测试结果显示：DolphinDB在5个查询性能测试中速度比MongoDB快40倍；在10个查询性能测试中速度比MongoDB快10~40倍；在6个查询性能测试中速度比MongoDB快数倍；仅在两个查询性能测试中，DolphinDB慢于MongoDB。

2、数据库导入和导出性能方面，测试结果显示：对于数据导入性能，DolphinDB的速度比MongoDB快大约130-460倍，且数据量越大，性能差距越大；对于数据输出性能，DolphinDB的速度比MongoDB快大约50倍。

3、数据库磁盘空间占用方面，测试结果显示：MongoDB数据库所占磁盘空间大约是DolphinDB数据库所占磁盘空间的4~6倍。

**2.环境配置**

本次测试在单机上进行，测试设备配置如下：

主机：DELL OptiPlex 7060

CPU：Intel(R) Core(TM) i7-8700 CPU@3.20GHZ，6核12线程

内存：32 GB (8GB x 4, 2666 MHz)

硬盘： 2T HDD (222MB/s读取；210MB/s写入)

OS：Ubuntu 18.04 LTS

DolphinDB选用Linux0.89作为测试版本，最大内存设置为28GB，线程数为12,所有节点最大连接数为128，分布式文件副本设置为1，设置1个控制节点，1个代理节点，3个数据节点。

MongoDB选用Linux4.0.5社区版作为测试版本，最大内存设置为28G，shard集群线程数为12，所有服务器的最大连接数均为128。MongoDB的shard集群设置为1个config服务器，1个mongos路由服务器，3个分片服务器，其中config服务器设置为有1个主节点和2个从节点的replica集群，3个分片服务器均设置为有1个主节点，1个从节点，1个仲裁节点的replica集群。DolphinDB和MongoDB的参数配置请参考附录1。

**3.数据集**

本报告测试了DolphinDB和MongoDB在小数据量级（4GB）和大数据量级（63GB）下的性能差距。

由于MongoDB社区版的未分区表在测试时会加入磁盘IO时间，导致MongoDB的查询时间增加，因此对于大小两种数据集，我们采用分区表，测试两种数据库在磁盘分区情况下的性能，查询时间均包含了磁盘IO的时间。为了保证测试的公平，我们在测试前通过linux命令：sync,echo1,2,3 | tee /proc/sys/vm/drop\_caches清空页面缓存，目录缓存和硬盘缓存，随后依次执行查询，并记录第一次执行的时间，11次查询前均未清空缓存。附录2是本次测试分区的脚本。

**4.2G设备传感器信息小数据集（CSV文件，3千万条数据）**

我们选用TimescaleDB官网提供的devices\_readings\_big.csv（以下简称readings数据集）和device\_info\_big.csv（以下简称info数据集）设备传感器数据作为小数据测试集。readings数据集包含3,000个设备在10,000个时间间隔（2016.11.15-2016.11.19）上的传感器信息，包括传感器时间，设备ID，电池，内存，CPU等时序统计信息。info数据集包括3,000个设备的设备ID，版本号，制造商，模式和操作系统等统计信息。

readings数据集有3千万条数据，大小为4GB，为一张设备传感器信息记录表；info数据集有3,000条数据，大小为100KB，为一张设备信息表。

数据来源：https://docs.timescale.com/v1.1/tutorials/other-sample-datasets

readings数据集结构如表1示，info数据集结构如表2所示。

**表1.readings数据集的结构**

|  |  |  |
| --- | --- | --- |
| **列名** | **DolphinDB数据类型** | **MongoDB数据类型** |
| time | DATETIME | Timestamp |
| device\_id | SYMBOL | String |
| battery\_level | INTEGER | Integer |
| battery\_status | SYMBOL | String |
| battery\_temperature | DOUBLE | Double |
| bssid | SYMBOL | String |
| cpu\_avg\_1min | DOUBLE | Double |
| cpu\_avg\_5min | DOUBLE | Double |
| mem\_free | DOUBLE | String |
| mem\_used | DOUBLE | Double |
| rssi | DOUBLE | Double |
| ssid | SYMBOL | String |

**表2.info数据集的结构**

|  |  |  |
| --- | --- | --- |
| **列名** | **DolphinDB数据类型** | **MongoDB数据类型** |
| device\_id | SYMBOL | String |
| api\_version | SYMBOL | String |
| manufacturer | SYMBOL | String |
| model | SYMBOL | String |
| os\_name | SYMBOL | String |

readings数据集的分区方式如下所示：

数据集中device\_id字段记录的是设备ID信息，总共3,000个不同值且重复出现，这种情况下用string类型不仅占用大量空间而且查询效率较低，在DolphinDB中使用symbol类型存储可以高效地解决存储空间和查询效率两个问题。readings数据集里面的time字段记录的是设备传感器从2016.11.15到2016.11.19总共4天的日期信息，device\_id字段为3,000个设备ID，数据集中其余字段大多为具体的传感器信息，这些字段的值变化较多且无规律，不确定性较高，分区后并不能起到减少搜索量的作用，因此并不适合分区。实际查询中我们往往根据设备和时间这两个字段来搜索具体某个设备在某个时间的运行情况，因此我们最终选择time和device\_id字段为分区字段。

分区字段确定后，根据实际查询的情况，我们在DolphinDB中采用组合分区方案，将time作为分区的第一维度，根据日期进行范围分区，每天分为一个区，总共分为四个区，再将设备ID作为第二维度，根据设备ID 进行范围分区，每天分为10个区，每个区包含300个设备。最后，总共分为40个区，每区大约为100MB。

我们在MongoDB中仍采用组合分区的方式，将time作为分区的第一维度，根据日期进行范围分区，再将设备ID作为第二分区维度，根据设备ID进行范围分区。MongoDB中的范围分区是根据块的大小进行范围分区的，当一个数据块大小大于某个阈值的时候，数据库会自动将一个大的数据块分为两个小的数据块，实现分区。经测试发现当chunksize（数据块分区阈值）为1,024的时候，性能最佳，最后，小数据集总共分为17个区。

MongoDB要求分区字段必须建立索引，因此我们建立日期+设备ID的复合索引，复合索引可以加快查询速度，但是MongoDB在建立索引时会消耗时间和空间。readings数据集分区建立time\_1\_device\_id\_1增序索引耗时为5分钟，占用空间大小为1.1G。建立索引的脚本如下所示：

use device\_pt

db.device\_readings.createIndex({time:1,device\_id:1}

对于测试中的结构化时序数据，DolphinDB无需建立索引就可以完成分区来加快查询速度，节省了建立索引的时间和空间。

**63G股票交易大数据集（CSV文件，16亿条数据）**

我们选用纽约证券交易所（NYSE）提供的2007.08.07-2007.08.10四天的股市Level1报价数据（以下简称TAQ数据集）作为大数据测试集，数据集包含8,000多支股票在4天内的交易时间，股票代码，买入价，卖出价，买入量，卖出量等报价信息。

数据集有4个csv文件，每个文件约16G，总共大小为63G，共16亿条数据，每个CSV文件保存一个交易日的交易信息，数据来源于(https://www.nyse.com/market-data/historical)。TAQ数据集结构如表3所示。

**表3.TAQ数据集的结构**

|  |  |  |
| --- | --- | --- |
| **列名** | **DolphinDB数据类型** | **MongoDB数据类型** |
| symbol | SYMBOL | String |
| date | DATEtime | Datetime |
| time | TIME | Datetime |
| bid | DOUBLE | Double |
| ofr | DOUBLE | Double |
| bidsiz | INT | Int |
| ofrsiz | INT | Int |
| mode | INT | Int |
| ex | CHAR | String |
| mmid | SYMBOL | String |

TAQ数据集的分区方式如下所示：

TAQ数据集里面的date字段记录的是设备传感器从2007.08.07到2007.08.10总共4天的日期信息，symbol字段记录的是股票代码信息，总共8,000多个不同值。实际查询中我们往往根据股票代码和时间进行查询，搜索某时间段内的某些股票的信息，这两个字段使用频率最高，因此我们选用date和symbol字段作为分区字段

我们在DolphinDB中采用组合分区方案，将date作为分区的第一维度，根据日期进行范围分区，每天分为一个区，总共分为四个区，可以快速定位到交易发生的时间，以减少查询搜索量，再将symbol作为第二维度，根据symbol 进行范围分区，每天分为100个区。最后，总共分为400个区，每区大约为40MB。

我们在MongoDB中仍采用组合分区的方式，将date作为分区的第一维度，根据日期进行范围分区，再将symbol作为第二分区维度。MongoDB中的范围分区情况和上述小数据集的一样，仍将chunksize设为1,024，总共分为385个区。

MongoDB在对TAQ数据集分区时建立date\_1\_symbol\_1增序索引消耗的时间为53分钟，占用空间大小为19G，建立索引的脚本如下所示：

use taq\_pt\_db

db.taq\_pt\_col.createIndex({date:1,symbol:1}

**4.数据库导入导出性能对比**

**从CSV文件导入数据**

在DolphinDB中使用以下脚本导入：

timer {

for (fp in fps) {

job\_id\_tmp = fp.strReplace(".csv", "")

job\_id\_tmp1=split(job\_id\_tmp,"/")

job\_id=job\_id\_tmp1[6]

job\_name = job\_id

submitJob(job\_id, job\_name, loadTextEx{db, `taq, `date`symbol, fp})

print now() + ": 已导入 " + fp

}

getRecentJobs(size(fps))

}

在DolphinDB中，readings数据集大小为4G，导入平均用时32秒，平均速率937,500条/秒。TAQ数据集大小为63G，导入平均用时432秒，平均速率3,703,700条/秒。

在MongoDB导入TAQ数据集时，为了加快导入速度，将63G的数据分为16个小文件导入，每个文件大小在3.5G~4.4G之间，然后使用以下脚本导入。

for f in /media/xllu/aa/TAQ/mongo\_split/\*.csv ; do

/usr/bin/mongoimport \

-h localhost \

--port 40000 \

-d taq\_pt\_db \

-c taq\_pt\_col \

--type csv \

--columnsHaveTypes \

--fields "symbol.string(),date.date(20060102),time.date(15:04:05),bid.double(),ofr.double(),bidsiz.int32(),ofrsiz.int32(),mode.int32(),ex.string(),mmid.string()" \

--parseGrace skipRow \

--numInsertionWorkers 12 \

--file $f

echo "文件 $f 导入完成"

done

在DolphinDB中，readings数据集大小为4G，导入平均用时74分钟，平均速率6,753条/秒。TAQ数据集大小为63G，导入平均用时3,302分钟，平均速率8,075条/秒。

**表4. 数据库导入的测试结果**

|  |  |  |  |
| --- | --- | --- | --- |
| **数据大小** | **MongoDB** | **DolphinDB** | **时长对比**  **MongoDB/DolphinDB** |
| 4G, 3千万条记录 | 6,753条/秒，共74分钟 | 937,500条/秒,共32秒 | 138 |
| 63G, 16亿条记录 | 8,075条/秒,  共3,302分钟 | 3,703,700条/秒，  共432秒 | 458 |

**测试结果分析：**

根据表4分析可得，DolphinDB在导入时序结构化数据时，速度远快于MongoDB。下面从横向和纵向两个方面来分析这次导入测试结果。

**1、纵向比较：**

导入readings数据集时，MongoDB的速度约为6,753条/秒，导入TAQ数据集时，速度约为8,075条/秒，这是因为MongoDB是按文档的格式来存储非结构化数据的，导入时要逐条进行插入。readings数据集的字段包含多个字符串类型，数值类型大多为double类型需要占据更大的空间。TAQ数据集的字段多为数值类型，其中int型和double型大约各占一半，而且字段数量还更少。相比于字符串类型，数值类型导入更快。因此可以看到MongoDB在导入TAQ大数据集时，速度反而更快。DolphinDB导入TAQ数据集时，导入速度加快也是由于字段类型不同和数量不同导致的。

**2、横向比较：**

MongoDB属于NoSQL数据库，采用文档形式存储数据，导入速度受文档数量的影响很大，可以看出导入3千万条记录大约需1小时，导入16亿条记录大约需55小时，记录条数相差大约53倍，导入时间相差约55倍，考虑到每条记录字段类型和数据的影响，可以认为导入速度和记录条数大约成正比。

DolphinDB属于列式内存关系型数据库，存储这种结构化的数据时，DolphinDB会将一个字段当成一个列，存储在一个列文件中，一个表有多少列，就有多少个列文件。导入readings数据集时创建了12个列文件，导入TAQ数据集时，创建了10个列文件，列文件数量相差不多。DolphinDB导入速度受文件大小影响较大，导入4G数据，需要32秒，导入63G数据，需要7分12秒。数据集大小相差大约15.75倍，导入时间相差大约13.5倍，考虑到列文件存储类型的不同，可以认为导入时间和文件大小大约成正比。

DolphinDB在导入时序结构化数据时，在列文件相差不大的情况下，导入时间和文件大小成正相关，符合列式存储数据库的特点。MongoDB在导入时序结构化数据时，在字段相差不大的情况下，导入时间和记录条数成正相关，符合文档型存储数据库的特点。

**3、MongoDB导入缓慢原因分析：**

对于MongoDB导入数据缓慢的原因，根据上文分析和资料查阅可以得到以下几点：

1、DolphinDB按照列式存储方式导入数据，在数据集列数增长不大的情况下，导入大数据受记录条数的影响不大。MongoDB按照记录逐条导入，在记录条数很大的情况下，MongoDB数据导入时长增加，性能下降。

2、MongoDB在sharding集群配置时，必须开启journaling日志，先写入记录再进行导入操作，降低了其导入速度。

3、因为MongoDB属于NoSQL数据库，其没有主键概念，为了保证唯一性约束，数据在导入时必须创建一个数据库自动生成的唯一索引来表征每一条记录，数据导入和索引必须同时进行，因此降低了导入速度。

**导出数据为CSV文件**

在DolphinDB中使用以下脚本进行数据导入：

timer saveText((select \* from t),"/media/xllu/aa/device/device\_readings\_out.csv")

在MongoDB中使用以下脚本进行数据导入：

mongoexport -h localhost:40000 -d db\_nopt -c device\_readings -o /media/xllu/aa/device/

device\_readings\_mongo\_out.csv

小数据集的导出性能如表5所示：

**表5. 数据库导出的测试结果**

|  |  |  |  |
| --- | --- | --- | --- |
| **数据大小** | **MongoDB** | **DolphinDB** | **时长对比**  **MongoDB/DolphinDB** |
| 4G, 3千万条记录 | 30,425条/秒，  共16分钟 | 1,500,000条/秒，  共20秒 | 49 |

**测试结果分析：**

因为数据导出跳过了数据类型判断和转换的过程，所以两个数据库导出过程都远快于导入过程，但DolphinDB按照列式存储结构导出，仅需导出40个列文件，导出速度仍比MongoDB快大约50倍。

综上所述，在导入和导出结构化数据时，DophinDB均快于MongoDB，并且数据记录越多，性能差距越明显。附录3是测试中导入和导出的脚本。

**5.数据库磁盘空间占用的性能对比**

数据库磁盘空间占用性能对比主要对比DolphinDB和MongoDB数据库导入readings数据集和TAQ数据集这两种大小的数据集后，在分区情况下各数据库中的数据所占磁盘空间的大小。磁盘占用指标为数据在磁盘中的大小。

DolphinDB直接通过读取所有列式数据文件的大小获取，MongoDB通过db.stats()获得数据存储的大小，MongoDB中数据存储大小还包括索引的大小。两个数据库均有一个副本集的备份，因此数据实际存储大小均为以下数值的一半。

**表6. 数据库磁盘空间占用的测试结果**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **MongoDB(G)** | **DolphinDB(G)** | **MongoDB/DolphinDB** |
| readings数据集 | 5.5 | 1.3 | 4.2 |
| TAQ数据集 | 82.3 | 11.9 | 6.9 |

**测试结果分析：**

根据表6分析可得，大数据集和小数据集的磁盘空间占用性能方面，DolphinDB均优于MongoDB，性能差距大约为5倍，且存储的数据越大，性能差距越大。主要有以下几个原因：

1. DolphinDB采用列式存储方式，每个列有固定的类型，通过LZ4压缩算法，将每个字段按照类型压缩存储为一个列文件，并且针对symbol类型，还采用位图压缩算法解决存储空间占用的问题，进一步提高了压缩率。
2. MongoDB中建立分区数据库均要对分区字段建立索引，进一步导致其存储空间变大，经分析发现，readings数据集对time和device\_id字段建立的索引大小为1.1G，TAQ数据集对date和symbol字段建立的索引大小为19G。
3. MongoDB本次测试采用的是WiredTiger存储引擎，选用snappy压缩算法，该算法的目标并不是最大限度地压缩数据而是在于提高压缩速度，因此压缩率不是很高。
4. MongoDB的存储单元是文档，当数据记录过多时，文档数量也特别多，其空间占用也会急剧上升，并且和文档数呈正相关性；DolphinDB为列式存储，只要列数增长不多，其空间占用增长也不会太大，并且和数据大小呈正相关性。

**6数据库查询的性能对比**

对于readings数据集和TAQ数据集，我们对比了以下8种常用的SQL查询。

1. 点查询：根据某一字段的具体值进行查询。
2. 范围查询：根据一个或者多个字段的范围根据时间区间进行查询。
3. 聚合查询：根据数据库提供的针对字段列进行计数，平均值，求和，最大值，最小值，标准差等聚合函数进行查询。
4. 精度查询：根据不同标签维度列进行数据聚合，实现高维或者低维的字段范围查询，测试有hour精度，minute精度。
5. 关联查询：根据不同的字段，在进行相同精度，相同的时间范围内进行过滤查询的基础上，筛选出有关联关系的指标列并进行分组。
6. 对比查询：根据两个维度将表中某字段的内容重新整理为一张表格（第一维度作为列，第二维度作为行）
7. 抽样查询：根据数据库提供的数据采样API，可以为每一次查询手动指定采样方式进行数据的稀疏处理，防止查询时间范围太大数据量过载的问题。
8. 经典查询：实际业务中常用的查询。

执行时间是以毫秒为单位的。为了消除网络传输等不稳定因素的影响，查询性能比较的时间指标为服务器执行某个查询的时间，不包括结果传输和显示的时间。

**4.2G设备传感器信息小数据集查询测试**

对于小数据集的测试，我们均测试磁盘分区数据，执行时间包括了磁盘IO的时间。为了保证测试的准确性和公正性，每次启动测试前均通过Linux系统命令sync；echo 1,2,3 | tee /proc/sys/vm/drop\_caches清除系统的页面缓存，目录项缓存和硬盘缓存，启动程序后一次执行样例一遍，并记录第一次执行完的时间。

DolphinDB中使用以下脚本得到数据库句柄：

login("admin","123456")

dp\_readings = "dfs://db\_range\_dfs"

device\_readings=loadTable(dp\_readings, `readings\_pt)

MongoDB中执行use device\_pt语句切换数据库至device\_pt数据库。

小数据集DolphinDB查询脚本如表7所示，小数据集MongoDB查询脚本详见附录4，查询性能测试结果如表8所示。

**表7 readings数据集性能测试脚本**

|  |  |  |
| --- | --- | --- |
| **ID** | **测试目的** | **DolphinDB查询脚本** |
| 1 | 点查询：按时间查询 | timer select \* from device\_readings  where time == 2016.11.15 07:00:00 |
| 2 | 点查询：根据设备ID查询 | timer select count(\*) from device\_readings  where device\_id = 'demo000101' |
| 3 | 范围查询.单分区维度：查询某时间段的所有记录 | timer select \* from device\_readings where time  between 2016.11.16 21:00:00 : 2016.11.17 21:30:00 |
| 4 | 范围查询.多分区维度：查询某时间段内某些设备的所有记录 | timer select \* from device\_readings where time between 2016.11.16 09:30:00 : 2016.11.17 09:30:00,device\_id in ['demo000001', 'demo000010', demo000100',demo001000'] |
| 5 | 范围查询.分区及非分区维度：查询某时间段内某些设备的特定记录 | timer select \* from device\_readings where time between 2016.11.15 20:00:00 : 2016.11.16 22:30:00,  device\_id in ['demo000002', 'demo000020', 'demo000200', 'demo002000'],battery\_level <= 50,battery\_status = 'discharging' |
| 6 | 精度查询：查询各设备每小时的最大内存使用 | timer select max(mem\_used) from device\_readings  group by hour(time) |
| 7 | 聚合查询.单维度分区max：计算各个设备的电池最大电量 | timer select max(battery\_level) from device\_readings  group by device\_id |
| 8 | 聚合查询.多维度分区avg：计算各时间段内设备电池平均温度 | timer select avg(battery\_temperature) from device\_readings  group by device\_id, hour(time) |
| 9 | 关联查询.多维过滤：查询某时间段内某些设备的特定信息 | timer select \* from lj(device\_readings,device\_info,`device\_id)  where time between 2016.11.15T09:00:00:2016.11.15T12:00:00,device\_id==`demo000100,battery\_status==`discharging |
| 10 | 关联查询.聚合查询：查询某段时间某些设备每小时的平均电池温度，按平均温度和设备排序 | timer select avg(battery\_temperature) as avg\_temperature,  std(cpu\_avg\_15min) as std\_cpu\_avg\_15min from lj(device\_readings,device\_info,`device\_id)  where device\_id between `demo000100:`demo000150,time between 2016.11.15T12:00:00:2016.11.16T12:00:00  group by hour(time),device\_id order by avg\_battery\_temperature desc,device\_id |
| 11 | 经典查询：计算某时间段内高负载高电量设备的内存大小 | timer select max(mem\_free) as mem\_all from evice\_readings  where time <= 2016.11.18 21:00:00,battery\_level >= 90,  cpu\_avg\_1min > 90 group by hour(time), device\_id |
| 12 | 经典查询：统计连接不同网络的设备的平均电量和最大、最小电量，并按平均电量降序排列 | timer select max(battery\_level) as max\_level,  avg(battery\_level) as avg\_level,min(battery\_level) as min\_level from device\_readings group by ssid order by avg\_battery desc |
| 13 | 经典查询：计算某个时间段内某些设备的总负载，并将时段按总负载降序排列 | timer select sum(cpu\_avg\_15min) as sum\_load from device\_readings where time between 2016.11.15 12:00:00 : 2016.11.16 12:00:00, device\_id in ['demo000001', 'demo000010', 'demo000100', 'demo001000']  group by hour(time) order by sum\_load desc |

**表8. readings数据集分区查询性能对比结果**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **测试内容** | **MongoDB**  **(ms)** | **DolphinDB**  **(ms)** | **MongoDB/**  **DolphinDB** |
| 1 | 点查询：按时间查询 | 616 | 3187 | 0.2 |
| 2 | 点查询：根据设备ID查询 | 80930 | 1506 | 53 |
| 3 | 范围查询.单分区维度：查询某时间段的所有记录 | 17956 | 8914 | 2 |
| 4 | 范围查询.多分区维度：查询某时间段内某些设备的所有记录 | 2542 | 170 | 15 |
| 5 | 范围查询.分区及非分区维度：查询某时间段内某些设备的特定记录 | 3763 | 112 | 33 |
| 6 | 精度查询：查询各设备每小时的最大内存使用 | 21460 | 1023 | 21 |
| 7 | 聚合查询.单维度分区max：计算各个设备的电池最大电量 | 22853 | 371 | 61 |
| 8 | 聚合查询.多维度分区avg：计算各时间段设备电池平均温度 | 17305 | 260 | 66 |
| 9 | 关联查询.多维过滤：查询某时间段内某些设备的特定信息 | 503 | 25 | 20 |
| 10 | 关联查询.聚合查询：查询某段时间某些设备每小时的平均电池温度，按平均温度和设备排序 | 2596 | 39 | 66 |
| 11 | 经典查询：计算某时间段内高负载高电量设备的内存大小 | 27408 | 1238 | 22 |
| 12 | 经典查询：统计连接不同网络的设备的平均电量和最大、最小电量，并按平均电量降序排列 | 19924 | 521 | 38 |
| 13 | 经典查询：计算某个时间段内某些设备的总负载，并将时段按总负载降序排列 | 51 | 5 | 10 |
|  | **中值** |  |  | 22 |

**测试结果分析：**

由上表分析可知：

**对于范围查询，**在包括了分区字段的查询中，如查询3，4所示，MongoDB可以调用复合索引，DolphinDB可以通过数据分区加快查询，这种情况下两个数据库的差距在15之内，并不是很大。在包括了未分区字段的查询中，如查询5所示，MongoDB没法调用未分区字段的索引，只能进行全表全字段搜索，DolphinDB则只进行未分区字段的单列搜索，这种情况下DolphinDB和MongoDB的差距扩大到30倍以上。可以看出在处理这种结构化时序数据时， DolphinDB采取的列式存储的方式的效率比MongoDB建立索引的方式更加高效，也更加适用于结构化数据的查询。

另一方面，当集合中的数据进行变化时，MongoDB需要随时维护索引，更新索引才能保证查询性能，这对于一些变化比较迅速的结构化时序数据来说，可操作性很差。DolphinDB则没有这方面的顾虑，通过高吞吐低延迟的列式内存引擎，DolphinDB可以高效地处理变化的时序数据。

**对于点查询**，在查询1中，MongoDB在建立有time+device\_id索引的情况下可以快速的找到某个时间点的记录，DolphinDB中按照日期分为4天，查找某一天的具体的时间点需要选定一个分区再进行检索，这种情况下DolphinDB比MongoDB慢。在查询2中，MongoDB的过滤字段仅为设备ID，没有包括time字段，我们从explain()中发现查询过程中没有调用复合索引，这是因为查询字段必须包括复合索引的首字段，索引才会起作用，查询1仅仅根据设备ID过滤，不涉及time字段的过滤， MongoDB不会调用复合索引，因此这种情况下，DolphinDB比MongoDB快。

MongoDB建立复合索引是因为在多字段查询中，复合索引的效率比多个单列索引更加高效，在实际查询情况，往往根据时间和设备来查找具体的信息，因此建立时间和设备的复合索引更加符合实际应用。如果仅仅根据设备ID过滤，需要额外地消耗时间和空间建立设备ID的单列索引或者是以设备ID 为首的复合索引。

**对于关联查询，**MongoDB作为NoSQL数据库，仅支持左外连接，并且因为其没有关系型数据库的主键约束，要实现表连接查询只能使用内嵌文档的方式，这种方式并不利于计算和聚合。DolphinDB作为关系型数据，其支持等值连接，左连接，全连接，asof连接，窗口连接和交叉连接，表连接查询功能丰富，可以高效方便地处理海量结构化时序数据。从查询9~10中我们可以看出DolphinDB快于MongoDB，并且关联查询越复杂，性能差距越大。

**对于抽样查询，**MongoDB可以在aggregation函数中通过$sample语句实现抽样查询，抽样方式取决于集合的大小，N（抽样数）的大小和$sample语句在pipeline中的位置。

1. 当$sample语句位于pipeline的第1位，并且N小于集合总记录的5%，并且集合包含100条以上的记录时，$sample语句会使用pseudo-random游标进行选择伪随机抽样选择。
2. 当上述三个条件有一个没有满足时，$sample语句会对全表进行扫描，排序并选取top N条记录，但是这样会受到MongoDB在排序时对于内存的限制，需要额外地设置allowDiskUse为true才能实现抽样功能。MongoDB进行抽样选择时，会将选择的记录的所有的字段都输出，详见MongoDB官网关于$sample的使用教程：https://docs.mongodb.com/manual/reference/operator/aggregation/sample/DolphinDB不支持全表抽样，仅支持分区字段抽样。因为两种数据抽样查询的实现过程差别较大，所以不做比较。

**对于插值查询，**MongoDB并没有内置的函数可以实现插值查询，而 DolphinDB 支持 4 种插值方式，ffill 向后取非空值填充、bfill 向前去非空值填充、lfill 线性插值、nullFill 指定值填充。

**对于对比查询，**MongoDB作为文档型数据库，其存储单元是集合，集合中包含若干文档，文档采用BSON格式，没有行和列的概念，因此无法实现选择两个维度将表中某字段的内容整理为一张表（第一个维度作为列，第二个维度作为行）的功能。DolphinDB中内置有pivot by函数语句，选定分类的维度可以方便的将制定内容整理为一张表。因为MongoDB不支持对比查询，所以不做比较。

**63G股票交易大数据集查询测试**

对于小数据集的测试，我们均测试磁盘分区数据，执行时间包括了磁盘IO的时间。为了保证测试的准确性和公正性，每次启动测试前均通过Linux系统命令sync；echo 1,2,3 | tee /proc/sys/vm/drop\_caches清除系统的页面缓存，目录项缓存和硬盘缓存，启动程序后一次执行样例一遍，并记录第一次执行完的时间。

DolphinDB中使用以下脚本得到数据库句柄：

login("admin","123456")

taq\_pt\_db= "dfs://db\_compound\_dfs"

taq\_pt\_col=loadTable(taq\_pt\_db, `readings\_pt)

MongoDB中执行use taq\_pt\_db语句切换数据库至taq\_pt\_db数据库。

大数据量DolphinDB的测试脚本如表9所示，大数据量MongoDB的测试脚本详见附录4，大数据量分区查询性能对比测试结果如表10所示。

**表9 TAQ数据集性能测试脚本**

|  |  |  |
| --- | --- | --- |
| **ID** | **测试目的** | **DolphinDB查询脚本** |
| １ | 点查询：按日期和股票代码查询 | timer select \* from taq\_pt\_col  where symbol = 'IBM', date == 2007.08.07 |
| 2 | 范围查询：查询某段时间内某些股票的所有信息 | timer select symbol, time, bid, ofr from taq\_pt\_col  where symbol in ('IBM', 'MSFT', 'GOOG', 'YHOO'),  date between 2007.08.07 : 2007.08.08,time between 09:30:00 : 09:40:00,bid > 20 |
| 3 | 范围查询：top 1000+排序，按股票代码,日期过滤，按ofr降序排序 | timer select top 1000 \*from taq\_pt\_col  where symbol in ('IBM', 'MSFT', 'GOOG', 'YHOO'),  date == 2007.08.07,time >= 07:36:37  order by ofr desc |
| ４ | 聚合查询.单分区维度：查询某天某个股票每分钟的最大ofr和最小的bid | timer select max(bid) as max\_bid,min(ofr) as min\_ofr  from taq\_pt\_col where date == 2007.08.10,  symbol == 'IBM',ofr > bidgroup by minute(time) |
| ５ | 聚合查询.多分区维度：查询某天某一些股票的bid标准差，按股票和分钟排序 | timer select std(bid) as std\_bid,sum(bidsiz) as sum\_bidsiz from taq\_pt\_col where date == 2007.08.07,time between 09:00:00 : 21:00:00,symbol in `IBM`MSFT`GOOG`YHOO,bid >= 20,ofr > 20  group by symbol, minute(time)  order by symbol asc, minute\_time asc |
| ６ | 经典查询：按股票代码，日期，时间，报价范围过滤，查询某些字段 | timer select symbol, time, bid, ofr from taq\_pt\_col　where symbol in ('IBM', 'MSFT', 'GOOG', 'YHOO'), date = 2007.08.08, time between 09:30:00 : 14:30:00, bid > 0, ofr > 10 |
| ７ | 经典查询：按日期，时间范围，bid范围，股票过滤，查询每个股票每分钟的bid标准差和平均值 | timer select std(bid) as std\_bid,avg(bid) as avg\_bid from taq\_pt\_col where date = 2007.08.09,time between 10:30:00 : 16:00:00,symbol in (`IBM,`MSFT,`GOOG,`YHOO),bid>1 group by symbol, minute(time) as minute |
| ８ | 经典查询：根据股票代码和日期过滤，统计买入和卖出的和的均值 | timer select max(ofr-bid) as max\_price　from taq\_pt\_col  where date = 2007.08.07,time between 07:30:00 : 10:00:00,symbol in ('GOOG',`SBW,'MSFT',`USBE,'YHOO')  group by symbol order by max\_price desc |
| ９ | 经典查询：按日期，时间，股票过滤，查询每天每分钟的均价 | timer select avg(ofr) as avg\_ofr,avg(bid) as avg\_bid　from　taq\_pt\_col where symbol = 'IBM', date between 2007.08.07 : 2007.08.08,time between 09:30:00 : 16:00:00 group by date, hour(time) as minute |
| 10 | 经典查询：按股票，时间，日期过滤，查询每只股票，每天的均价 | timer select avg(ofr + bid) as avg\_price from taq\_pt\_col  where date between 2007.08.08 : 2007.08.09,time between 12:00:00 : 17:30:00,symbol in (`IPB,`SBW,`IBM,`USBE,'YHOO') group by symbol, date |

**表10. TAQ数据集分区查询性能对比结果**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **测试内容** | **MongoDB**  **(ms)** | **DolphinDB**  **(ms)** | **MongoDB/**  **DolphinDB** |
| 1 | 点查询：按日期和股票代码查询 | 536 | 542 | 0.98 |
| 2 | 范围查询：查询某段时间内某些股票的所有信息 | 9468 | 1466 | 6 |
| 3 | 范围查询：top 1000+排序，按股票代码,日期过滤，按ofr降序排序 | 4057 | 452 | 9 |
| 4 | 聚合查询.单分区维度：查询某天某个股票每分钟的最大ofr和最小的bid | 3134 | 345 | 9 |
| 5 | 聚合查询.多分区维度：查询某天某一些股票的bid标准差，按股票和分钟排序 | 1629 | 60 | 27 |
| 6 | 经典查询：按股票代码，日期，时间，报价范围过滤，查询某些字段 | 3658 | 128 | 28 |
| 7 | 经典查询：按日期，时间范围，bid范围，股票过滤，查询每个股票每分钟的bid标准差和平均值 | 11752 | 968 | 12 |
| 8 | 经典查询：根据股票代码和日期过滤，统计买入和卖出的和的均值 | 1996 | 329 | 6 |
| 9 | 经典查询：按日期，时间，股票过滤，查询每天每分钟的均价 | 2821 | 55 | 45 |
| 10 | 经典查询：按股票，时间，日期过滤，查询每只股票，每天的均价 | 4122 | 984 | 4 |
|  | **中值** |  |  | 9 |

**测试结果分析：**

根据表5分析可知：在大数据量，两个数据库均做了分区的情况下，DolphinDB依然比MongoDB大约快10-45倍。查询1是根据日期，股票代码进行的点查询，这种情况下MongoDB和DolphinDB的性能差距不大，DolphinDB略慢于MongoDB，这是MongoDB进行复合分区时会建立date+symbol复合索引可以较为快速的找到结果，是MongoDB比较好的应用场景，但是对比查询5～9可知，MongoDB的计算性能仍不如DolphinDB，差距大约在25-45倍之间。MongoDB在对TAQ数据集分区时建立date\_1\_symbol\_1增序索引消耗的时间为53分钟，占用空间大小为19G，因为需要维护索引，仍然不适用于高频变化的时序数据。

**8.结论**

综上所述，DolphinDB在处理时序结构化数据时，在查询速度，空间占用，数据导入和导出方面性能都更加优越，大多数情况下，查询速度大约是MongoDB的10~40倍，数据导入速度大约是MongoDB的130～460倍，数据导出速度大约是MongoDB的50倍。

MongoDB在处理时序结构化数据时对索引的依赖较大，很多查询性能都依赖索引的建立，但是建立索引的方式并不适用于需要频繁进行插入，删除和更新的数据集，因为对数据集进行改动后需要额外地进行索引的更新和维护，这对于变化频率比较快的数据集而言，可操作性比较差。DolphinDB查询性能不依赖索引，通过高效灵活的分区方案和高吞吐低延迟的列式内存引擎，可以处理高频变化的时序结构化数据时，具有很大的优势。

除此之外，DolphinDB还具有以下优势：

1、DolphinDB的查询语言兼容SQL语言，操作命令大致相同，编程语言和Python类似，数据结构丰富，大大降低了学习成本，相比之下MongoDB的语言较为晦涩，操作人员需要从头学习MongoDB的查询语言，才能熟练操作，学习成本较高。

2、DolphinDB底层优化更加突出，配备分布式数据库管理系统，查询和计算功能丰富，可以胜任大数据量情况下的复杂操作。

**附录1：环境配置**

**1.1、DolphinDB环境配置**

**1.1.1、controller.cfg**

localSite=localhost:8920:ctl8920

localExecutors=11

maxConnections=128

maxMemSize=2

webWorkerNum=12

workerNum=12

dfsReplicationFactor=1

dfsReplicaReliabilityLevel=0

**1.1.2、agent.cfg**

workerNum=12

localExecutors=11

maxMemSize=2

localSite=localhost:8910:agentgetJobStat()

controllerSite=localhost:8920:ctl8920

**1.1.3、cluster.cfg**

maxConnections=128

maxMemSize=8

workerNum=12

localExecutors=11

webWorkerNum=12

**1.1.4、cluster.nodes**

localSite,mode

localhost:8910:agent,agent

localhost:8080:local8080,datanode

localhost:8081:local8081,datanode

localhost:8082:local8082,datanode

**1.2、Mongodb环境配置**

**1.2.1、MongoDB配置服务器设置**

## 配置文件内容

pidfilepath = /media/xllu/aa/localhost\_shard\_jiqun/conf/conf\_m/conf\_m.pid

dbpath = /media/xllu/aa/localhost\_shard\_jiqun/conf/conf\_m/conf\_m\_data

logpath = /media/xllu/aa/localhost\_shard\_jiqun/conf/conf\_m/conf\_m.log

directoryperdb=true

logappend = true

bind\_ip = localhost

port = 20000

#以后台方式启动

fork = true

#declare this is a config db of a cluster;

configsvr = true

replSet=conrep

noprealloc=true

#设置最大连接数

maxConns=128

wiredTIgerCacheSizeGB=2

**1.2.2、MongoDB路由服务器设置**

nohup ./mongos --configdb conrep/localhost: 20000,localhost:30001,localhost:30002 --bind\_ip localhost --port 40000 --logpath /media/xllu/aa/localhost\_shard\_jiqun/mongos/mongos40000.log --logappend --setParameter ShardingTaskExecutorPoolHostTimeoutMS=30000000 --setParameter taskExecutorPoolSize=6 --setParameter ShardingTaskExecutorPoolMaxConnecting=12 &

**1.2.3、MongoDB分片服务器设置**

## 配置文件内容

pidfilepath = /media/xllu/aa/localhost\_shard\_jiqun/shard1/shard1\_m/shard1\_m.pid

dbpath = /media/xllu/aa/localhost\_shard\_jiqun/shard1/shard1\_m/shard1\_m\_data

logpath = /media/xllu/aa/localhost\_shard\_jiqun/shard1/shard1\_m/shard1\_m.log

directoryperdb=true

logappend = true

bind\_ip = localhost

port = 20001

fork = true

#declare this is a shard db of a cluster;

shardsvr = true

#副本集名称

replSet=shardrep1

noprealloc=true

#设置最大连接数

maxConns=128

wiredTIgerCacheSizeGB=4

**2.1、Dolphindb样本生成脚本**

**2.1.1、device\_readings.csv数据集分区脚本getJobStat()**

login("admin","123456")

//导入csv文件为内存表

fp\_info = '/media/xllu/aa/device/devices\_info.csv'

fp\_readings = '/media/xllu/aa/device/devices\_readings.csv'

dp\_readings = "dfs://db\_range\_dfs"

//dp\_readings = "/home/xllu/database/db\_range"

// 创建两张表的 schema

colnames\_info = `device\_id`api\_version`manufacturer`model`os\_name

colnames\_readings = `time`device\_id`battery\_level`battery\_status`battery\_temperature`bssid`cpu\_avg\_1min`cpu\_avg\_5min`cpu\_avg\_15min`mem\_free`mem\_used`rssi`ssid

types\_info = `SYMBOL`INT`STRING`STRING`STRING

types\_readings = `DATETIME`SYMBOL`INT`STRING`DOUBLE`STRING`DOUBLE`DOUBLE`DOUBLE`DOUBLE`DOUBLE`DOUBLE`SYMBOL

schema\_info = table(colnames\_info, types\_info)

schema\_readings = table(colnames\_readings, types\_readings)

time\_range=2016.11.15T00:00:00 + 86400 \* 0..4

id\_range=('demo'+lpad((0..10\*300)$STRING,6,"0"))$SYMBOL

time\_schema=database('',RANGE,time\_range)

id\_schema=database('',RANGE,id\_range)

// 创建分区数据库并定义分区方式

if(existsDatabase(dp\_readings)){

dropDatabase(dp\_readings)

}

db = database(dp\_readings , COMPO, [time\_schema,id\_schema])

// 从 CSV 导入 readings 表的数据到 readings 数据库并完成数据分区操作

timer loadTextEx(db, `readings\_pt, `time`device\_id, fp\_readings, , schema\_readings)

//38s

// 从 CSV 导入 device\_info 表的数据到 device\_info 内存表

device\_info = loadText(fp\_info, , schema\_info)

// 加载 readings 表（不包括表连接）

readings = loadTable(db, `readings\_pt)

**2.1.2、TAQ20070807.csv-TAQ20070810.csv数据集分区脚本**

login("admin","123456")

FP\_TAQ = '/media/xllu/aa/TAQ/original/'device\_

FP\_SAMPLE\_TB = FP\_TAQ + 'TAQ20070807.csv'

orig\_tb\_schema = extractTextSchema(FP\_SAMPLE\_TB)

// 查看 orig\_tb\_schema

// 将列名调整为小写避免与 DolphinDB 内置的 SYMBOL, DATE, TIME 等保留关键字产生冲突

cols = lower(orig\_tb\_schema.name)

schema = table(cols, orig\_tb\_schema.type)

timer sample\_tb = loadText(FP\_SAMPLE\_TB, , schema)

sample\_freq\_tb = select count(\*) from sample\_tb group by symbol

mmid\_tb = select count(\*) from sample\_tb group by mmid

// 8369 rows, [symbol, count], 分到 100 个 buckets

BIN\_NUM = 100

buckets = cutPoints(sample\_freq\_tb.symbol, BIN\_NUM, sample\_freq\_tb.count)

// [A, ABL, ACU, ..., ZZZ], 101 个边界

buckets[BIN\_NUM] = `ZZZZZZ // 调整最右边界

DATE\_RANGE = 2007.08.05..2007.08.12

// 创建数据库分区方案

date\_schema = database('', VALUE, DATE\_RANGE)

symbol\_schema = database('', RANGE, buckets)

//FP\_DB = FP\_TAQ + 'db/'

//db = database(FP\_DB, COMPO, [date\_schema, symbol\_schema])

db\_path="dfs://db\_compound\_dfs"

if(existsDatabase(db\_path)){

dropDatabase(db\_path)

}

db = database(db\_path, COMPO, [date\_schema, symbol\_schema])

fps = FP\_TAQ + (exec filename from files(FP\_TAQ) order by filename)

// 导入到分布式数据库中

timer {getJobStat()

for (fp in fps) {

job\_id\_tmp = fp.strReplace(".csv", "")

job\_id\_tmp1=split(job\_id\_tmp,"/")

job\_id=job\_id\_tmp1[6]

job\_name = job\_id

submitJob(job\_id, job\_name, loadTextEx{db, `taq, `date`symbol, fp})

print now() + ": 已导入 " + fp

}

getRecentJobs(size(fps))

}

**2.2、Mongodb样本生成脚本**

**2.2.1、readings数据集分区脚本getJobStat()**

use config

db.chunks.find()

db.settings.save({"\_id":"chunksize","value":1024})

use admin

sh.enableSharding("device\_pt")

Use device\_pt

db.device\_readings.createIndex({time:1,device\_id:1})

//为节约导入时间，我们先导入数据，在mongoimport文件目录下输入以下脚本

./mongoimport -h localhost:40000 -d db\_nopt -c device\_readings --file /media/xllu/aa/device/device\_readings\_mongo.csv --type csv --columnsHaveTypes --fields "time.date(2006-01-02T15:04:05),device\_id.string(),battery\_level.int32(),battery\_status.string(),battery\_temperature.double(),bssid.string(),cpu\_avg\_1min.double(),cpu\_avg\_5min.double(),cpu\_avg\_15min.double(),mem\_free.double(),mem\_used.double(),rssi.double(),ssid.string()" --parseGrace skipRow --numInsertionWorkers 12

//回到shell界面，输入以下脚本完成分区

sh.shardCollection("device\_pt.device\_readings",{"time":1,"device\_id":1})

**2.2.2、TAQ数据集分区脚本**

use config

db.chunks.find()

db.settings.save({"\_id":"chunksize","value":1024})

use admin

sh.enableSharding("taq\_pt\_db")

Use device\_pt

db.device\_readings.createIndex({date:1,symbol:1})

/\*为节约导入时间，我们将数据分为16个小数据表进行分别导入，在shell中输入以下bash脚本\*/

#!/bin/bash

for f in /media/xllu/aa/TAQ/mongo\_split/\*.csv ; do

/usr/bin/mongoimport \

-h localhost \

--port 40000 \

-d taq\_pt\_db \

-c taq\_pt\_col \

--type csv \

--columnsHaveTypes \

--fields "symbol.string(),date.date(20060102),time.date(15:04:05),bid.double(),ofr.double(),bidsiz.int32(),ofrsiz.int32(),mode.int32(),ex.string(),mmid.string()" \

--parseGrace skipRow \

--numInsertionWorkers 12 \

--file $f

echo "文件 $f 导入完成"这个

done

//回到shell界面，输入以下脚本完成分区

sh.shardCollection("device\_pt.device\_readings",{"time":1,"device\_id":1})

**附录3：数据库输入输出的性能对比脚本**

DolphinDB导入csv文件脚本：

导入readings数据集

timer loadTextEx(db, `readings\_pt,`time, datapath, ,schema\_readings)

导入TAQ数据集

timer {

for (fp in fps) {

job\_id\_tmp = fp.strReplace(".csv", "")

job\_id\_tmp1=split(job\_id\_tmp,"/")

job\_id=job\_id\_tmp1[6]

job\_name = job\_id

submitJob(job\_id, job\_name, loadTextEx{db, `taq, `date`symbol, fp})

print now() + ": 已导入 " + fp

}

getRecentJobs(size(fps))

}

MongoDB导入csv文件脚本：

导入readings数据集

./mongoimport -h localhost:40000 -d db\_nopt -c device\_readings --file /media/xllu/aa/device/

device\_readings\_mongo.csv --type csv --columnsHaveTypes --fields "time.date(2006-01-02T15:04:05),device\_id.string(),battery\_level.int32(),battery\_status.string(),battery\_temperature.double(),bssid.string(),cpu\_avg\_1min.double(),cpu\_avg\_5min.double(),cpu\_avg\_15min.double(),mem\_free.double(),mem\_used.double(),rssi.double(),ssid.string()" --parseGrace skipRow --numInsertionWorkers 12

导入TAQ数据集

for f in /media/xllu/aa/TAQ/mongo\_split/\*.csv ; do

/usr/bin/mongoimport \

-h localhost \

--port 40000 \

-d taq\_pt\_db \

-c taq\_pt\_col \

--type csv \

--columnsHaveTypes \

--fields "symbol.string(),date.date(20060102),time.date(15:04:05),bid.double(),ofr.double(),bidsiz.int32(),ofrsiz.int32(),mode.int32(),ex.string(),mmid.string()" \

--parseGrace skipRow \

--numInsertionWorkers 12 \

--file $f

echo "文件 $f 导入完成"

done

DolphinDB导出csv文件脚本：

timer saveText((select \* from t),"/media/xllu/aa/device/device\_readings\_out.csv")

MongoDB导出csv文件脚本：

mongoexport -h localhost:40000 -d db\_nopt -c device\_readings -o /media/xllu/aa/device/

device\_readings\_mongo\_out.csv

**附录4：用于数据库查询性能对比的脚本**

**4.1、readings数据集性能测试脚本**

|  |  |  |
| --- | --- | --- |
| **ID** | **测试目的** | **MongoDB查询脚本** |
| 1 | 点查询：按时间查询 | db.device\_readings.explain("executionStats").find(  {time:{"$eq":ISODate("2016-11-15 07:00:00.000Z")}},{}) |
| 2 | 点查询：根据设备ID查询 | db.device\_readings.explain("executionStats").find(  {device\_id:{"$eq":"demo000101"}},{}).count() |
| 3 | 范围查询：单分区维度：查询某时间段的所有记录 | db.device\_readings.explain("executionStats").find(  {time:{"$gte":ISODate("2016-11-16 21:00:00.000Z"),"$lte":ISODate("2016-11-17 21:30:00.000Z")}},{}) |
| 4 | 范围查询.多分区维度：查询某时间段内某些设备的所有记录 | db.device\_readings.explain("executionStats").find(  {time:{"$gte":ISODate("2016-11-16 09:30:00.000Z"),"$lte":ISODate("2016-11-17 09:30:00.000Z")},  device\_id:{"$in":["demo000001","demo000010","demo000100","demo001000"]}},{}) |
| 5 | 范围查询.多分区维度：查询某时间段内某些设备的特定记录 | db.device\_readings.explain("executionStats").find(  {time:{"$gte":ISODate("2016-11-15 20:00:00.000Z"),"$lte":ISODate("2016-11-16 22:30:00.000Z")},  device\_id:{"$in":["demo000002","demo000020","demo000200","demo002000"]},battery\_level:{"$lte":50},battery\_status:"discharging"},{}) |
| 6 | 精度查询：查询各设备每小时的最大内存使用 | db.device\_readings.explain("executionStats").aggregate([{$group:{\_id:{hour\_new:{$hour:"$time"}},max\_mem:{$max:"$mem\_used"}}}]) |
| 7 | 聚合查询.单维度分区max：计算各设备电池最大电量 | db.device\_readings.explain("executionStats").aggregate([{$group:{\_id:{device\_id:"$device\_id",hour\_new:{$hour:"$time"}},max\_temperature:{$max:"$battery\_temperature"}}}]) |
| 8 | 聚合查询.多维度分区avg：计算各时间段内设备电池平均温度 | db.device\_readings.explain("executionStats").aggregate([{$group:{\_id:{device\_id:"$device\_id",},  avg\_temperature:{$avg:"$battery\_temperature"}}}]) |
| 9 | 关联查询.多维过滤：查询某时间段内某些设备的特定信息 | db.device\_readings.explain("executionStats").aggregate([{$lookup:{from:"device\_info",localField:"device\_id", foreignField:"device\_id",as:"device\_result"}},  {$match:{time:{"$gte":ISODate("2016-11-15 09:00:00.000Z"),"$lte":ISODate("2016-11-15 12:00:00.000Z")},device\_id:{"$eq":"demo000100"},  battery\_status:"discharging"}}]) |
| 10 | 关联查询.聚合查询：查询某段时间某些设备每小时的平均电池温度，按平均温度和设备排序 | db.device\_readings.explain("executionStats").aggregate([{$lookup:{from:"device\_info",localField:"device\_id",foreignField:"device\_id",as:"device\_result"}},{$match:{device\_id:{"$gte":"demo000100","$lte":"demo000150"},time:{"$gte":ISODate("2016-11-15 12:00:00.000Z"),"$lte":ISODate("2016-11-16 12:00:00.000Z")},cpu\_avg\_15min:{"$gte":5}}},  {$group:{\_id:{hour\_new:{$hour:"$time"},device\_id:"$device\_id"},avg\_battery\_temperature:{$avg:"$battery\_temperature"},std\_cpu\_avg\_15min:{$stdDevSamp:"$cpu\_avg\_15min"}}},{$sort:{"avg\_battery\_temperature":-1,"device\_id":1}}]) |
| 11 | 经典查询：计算某时间段内高负载高电量设备的内存大小 | db.device\_readings.explain("executionStats").aggregate([{$match:{time:{"$lte":ISODate("2016-11-18 21:00:00.000Z")},battery\_temperature:{"$gte":90},cpu\_avg\_1min:{"$gte":90}}},{$group:{\_id:{time:"$time",device\_id:"$device\_id"},max\_mem\_free:{$max:"$mem\_free"}}}]) |
| 12 | 经典查询：统计连接不同网络的设备的平均电量和最大、最小电量，并按平均电量降序排列 | db.device\_readings.explain("executionStats").aggregate([{$group:{\_id:{ssid:"$ssid"},std\_battery\_level:{$stdDevSamp:"$battery\_level"},avg\_battery\_level:{$avg:"$battery\_level"}}},  {$sort:{"avg\_battery\_level":-1}}]) |
| 13 | 关联查询：计算某个时间段内某些设备的总负载，并将时段按总负载降序排列 | db.device\_readings.explain("executionStats").aggregate([{$match:{time:{"$gte":ISODate("2016-11-15 12:00:00.000Z"),"$lte":ISODate("2016-11-16 12:00:00.000Z")}, device\_id:{"$in":["demo000001","demo000010","demo000100","demo001000"]}}},{$group:{\_id:{hour\_new:{$hour:"$time"}},sum\_15min:{$sum:"$cpu\_avg\_15min"}}},{$sort:{"sum\_15min":-1}}]) |

**4.2、TAQ数据集性能测试脚本**

|  |  |  |
| --- | --- | --- |
| **ID** | **测试目的** | **MongoDB查询脚本** |
| １ | 点查询：按日期和股票代码查询 | db.taq\_pt\_col.explain("executionStats").find({symbol:{"$eq":"IBM"},date:{"$eq":ISODate("2007-08-07T00:00:00.000Z")}}) |
| 2 | 范围查询：查询某段时间内某些股票的所有信息 | db.taq\_pt\_col.explain("executionStats").find({symbol:{"$in":["IBM","MSFT","GOOG","YHOO"]},date:{"$gte":ISODate("2007-08-07T00:00:00.000Z"),"$lte":ISODate("2007-08-08T00:00:00.000Z")},time:{"$gte":ISODate("0000-01-01T09:30:00.000Z"),"$lte":ISODate("0000-01-01T09:40:00.000Z")},bid:{"$gt":20}},{symbol:1,time:1,bid:1,ofr:1}) |
| 3 | 范围查询：top 1000+排序，按股票代码,日期过滤，按ofr降序排序 | db.taq\_pt\_col.explain("executionStats").aggregate([{$match:{symbol:{"$in":["IBM","MSFT","GOOG","YHOO"]},date:{"$eq":ISODate("2007-08-07T00:00:00.000Z")},time:{"$gte":ISODate("0000-01-01T07:36:37.000Z")}}},{$sort:{ofr:-1}},{$limit:1000}]) |
| ４ | 聚合查询.单分区维度：查询某天某个股票每分钟的最大ofr和最小的bid | db.taq\_pt\_col.explain("executionStats").aggregate([{$project:{symbol:1,date:1,bid:1,ofr:1,time:1,ofr\_bid\_dif:{$subtract:["$ofr","$bid"]}}},{$match:{date:{"$eq":ISODate("2007-08-10T00:00:00.000Z")},  symbol:{"$eq":"IBM"},ofr\_bid\_dif:{"gt":0}}},{$group:  {\_id:{minute\_new:{$minute:"$time"}},max\_bid:{$max:"$bid"},min\_ofr:{$min:"$ofr"}}}]) |
| ５ | 聚合查询.多分区维度：查询某天某一些股票的bid标准差，按股票和分钟排序 | db.taq\_pt\_col.explain("executionStats").aggregate([  {$match:{date:{"$eq":ISODate("2007-08-07T00:00:00.000Z")},time:{"$gte":ISODate("0000-01-01T09:00:00.000Z"),"$lte":ISODate("0000-01-01T21:00:00.000Z")},symbol:{"$in":["IBM","MSFT","GOOG","YHOO"]},bid:{"gt":20},ofr:{"gt":20}}},{$group:{\_id:{symbol:"$symbol",minute\_new:{$minute:"$time"}},std\_bid:{$stdDevSamp:"$bid"},sum\_bidsiz:{$sum:"$bidsiz"}}},{$sort:{symbol:1,minute\_new:1}}]) |
| ６ | 经典查询：按股票代码，日期，时间，报价范围过滤，查询某些字段 | db.taq\_pt\_col.explain("executionStats").find(  {symbol:{"$in":["IBM","MSFT","GOOG","YHOO"]},date:{"$eq":ISODate("2007-08-08T00:00:00.000Z")},  time:{"$gte":ISODate("0000-01-01T09:30:00.000Z"),"$lte":ISODate("0000-01-01T14:30:00.000Z")},bid:{"$gt":0},ofr:{"$gt":10}},{"symbol":1,"time":1,"bid":1,"ofr":1}) |
| ７ | 经典查询：按日期，时间范围，bid范围，股票过滤，查询每个股票每分钟的bid标准差和平均值 | db.taq\_pt\_col.explain("executionStats").aggregate([{$match:{date:{"$eq":ISODate("2007-08-09T00:00:00.000Z")},time:{"$gte":ISODate("0000-01-01T10:30:00.000Z"),"$lte":ISODate("0000-01-01T16:00:00.000Z")},symbol:{"$in":["IBM","MSFT","GOOG","YHOO"]},bid:{"gt":1}}},{$group:{\_id:{symbol:"$symbol",minute\_new:{$minute:"$time"}},std\_bid:{$stdDevSamp:"$bid"},avg\_bid:{$avg:"$bid"}}}]) |
| ８ | 经典查询：根据股票代码和日期过滤，统计买入和卖出的和的均值 | db.taq\_pt\_col.explain("executionStats").aggregate([{$project:{symbol:1,date:1,bid:1,ofr:1,time:1,ofr\_bid\_dif:{$subtract:["$ofr","$bid"]}}},{$match:{date:{"$eq":ISODate("2007-08-07T00:00:00.000Z")},  time:{"$gte":ISODate("0000-01-01T07:30:00.000Z"),"$lte":ISODate("0000-01-01T10:00:00.000Z")},symbol:{"$in":["GOOG","SBW","MSFT","USBE","YHOO"]}}},{$group:{\_id:{symbol:"$symbol"},max\_price:{$max:"$ofr\_bid\_dif"}}},{$sort:{max\_price:-1}}]) |
| ９ | 经典查询：按日期，时间，股票过滤，查询每天每分钟的均价 | db.taq\_pt\_col.explain("executionStats").aggregate([{$match:{symbol:{"$eq":"IBM"},date:{"$gte":ISODate("2007-08-07T00:00:00.000Z"),"$lte":ISODate("2007-08-08T00:00:00.000Z")},time:{"$gte":ISODate("0000-01-01T09:30:00.000Z"),"$lte":ISODate("0000-01-01T16:00:00.000Z")}}},{$group:{\_id:{date\_new:{$dayOfMonth:"$time"},minute\_new:{$minute:"$time"}},avg\_bid:{$avg:"$bid"},avg\_ofr:{$avg:"$ofr"}}}]) |
| 10 | 经典查询：按股票，时间，日期过滤，查询每只股票，每天的均价 | db.taq\_pt\_col.explain("executionStats").aggregate([  {$project:{symbol:1,date:1,time:1,ofr\_bid\_sum:{$add:["$ofr","$bid"]}}},{$match:{date:{"$gte":ISODate("2007-08-08T00:00:00.000Z"),"$lte":ISODate("2007-08-09T00:00:00.000Z")},time:{"$gte":ISODate("0000-01-01T12:00:00.000Z"),"$lte":ISODate("0000-01-01T17:30:00.000Z")},symbol:{"$in":["IPB","SBW","IBM","USBE","YHOO"]}}},{$group:{\_id:{symbol:"$symbol",date\_new:{$dayOfMonth:"$time"}},avg\_price:{$avg:"$ofr\_bid\_sum"}}}]) |