Database Lab4

Lilin Wang & Yanjing Zhang

GPS trajectory data(Using 5 sets)

MySql

Storing Data

We store each measure as a row into a table Trajectory, the table's structure is like following,

```
CREATE TABLE Trajectory (
        SetId char(5) not NULL,
        TrajectoryID char(15) not NULL,
         lat double(12,8),
        lon double(12,8),
        alt int(6),
        datenum double(18,11),
        date char(11),
        time char(10) )
```

Querying Date

1. Count the number of GPS points in the trajectory.

   Return the count of rows whose "SetID" equals provided setid and "TrajectoryID" equals defined trajectoryid

2. Select 10 different days, and for each day, find the number of GPS points that were measured on that day.

   Return the count of rows whose "date" equals provided date

Time consumption

Case/ Time(ns)	1	2	3	4	5
Insert into DB	351103803868	341706028960	344096003484	342370189447	337376933776
Count number	587446023	578168005	607320547	602797468	610619868
Count date	4802387067	479220113	4956240568	4921631698	4961183987

Document store: MongoDB

Storing Data

We store each measure as a document into MongoDB, the structure is like following,

{ "_id" : ObjectId("547418f1300452579430025f"), "TrajectorySetID" : "000", "TrajectoryID" : "20081023025304", "Latitude" : 39.98447, "Longtitude" : 116.308827, "Altitude" : 257, "Date" : 39744.1236111111 }

Querying Date

1. Count the number of GPS points in the trajectory.
   Return the count of documents whose "TrajectorySetID" equals provided setid and "TrajectoryID" equals defined trajectoryid

2. Select 10 different days, and for each day, find the number of GPS points that were measured on that day.

Return the count of documents whose "Date" equals provided date

Time consumption

Case/ Time(ns)	1	2	3	4	5
Insert into DB	127641202966	125697966882	125222846892	129169023263	129743884644
Count number	45530692	47120451	48060204	46763818	47586373
Count date	5989851636	6190969990	6161586115	6187229852	6232348071

Database Lab4

Lilin Wang & Yanjing Zhang

Key-value: Redis

Storing Data

We store each Trajectory as a list(SetID+TrajectoryID as key) into Redis, and each measure is an element of a list, the structure is like following,

> *Key: "00020081023025304"*
> *Data: {"40.002684,116.323946,959,39986.4128703704,2009-06-22,09:54:32"}, {"40,116.325545,105,39986.4131597222,2009-06-22,09:54:57"}, ...*

Querying Date

1. Count the number of GPS points in the trajectory.

   Return the size of the list whose Key equals provided setid + trajectoryid

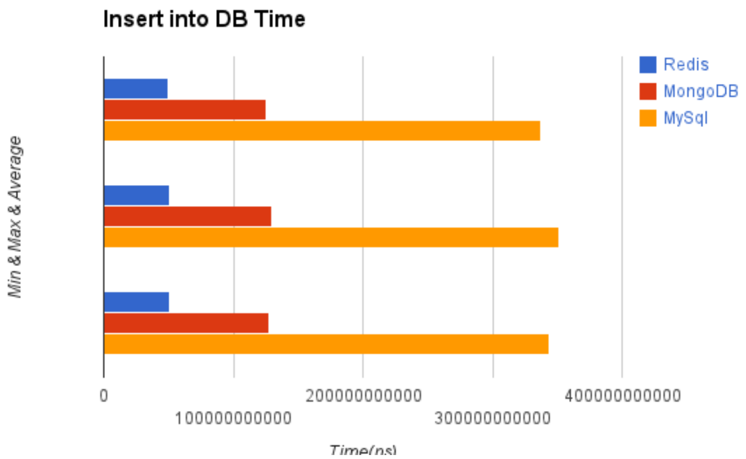
2. Select 10 different days, and for each day, find the number of GPS points that were measured on that day.

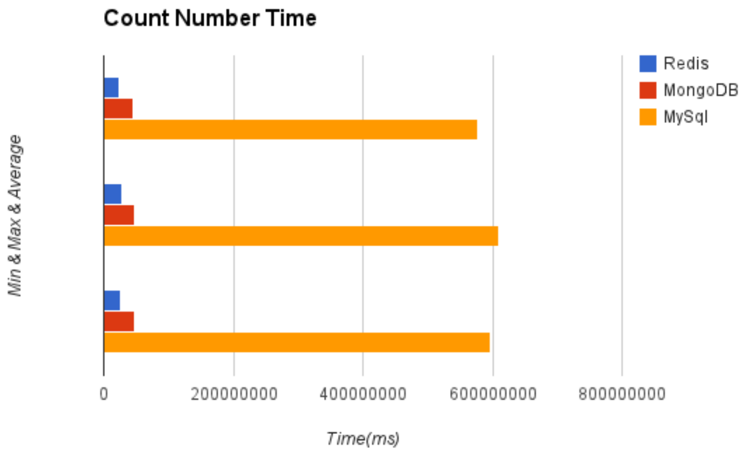
   Implemented by two loops, first traverse all lists, and then traverse all measures in the list. In each measure check if the date filed equals provided date, if yes, count++.

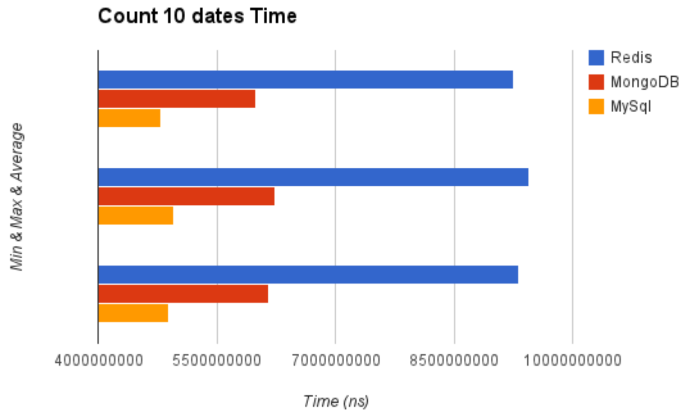
Time consumption

Case/ Time(ns)	1	2	3	4	5
Insert into DB	50121956950	51083670161	51190599183	50121956950	50488010953
Count number	24426743	24017282	26549638	23873036	27185598
Count date	9292069671	9252741494	9328643413	9279856441	9456104037

Comparison







Summary:
1. Create DB speed: Redis>MongoDb>MySql
2. When using Redis to query count of 10 days, we have to use two loops to traverse all measures, so it's the slowest one.

3. When using select count(*) from X in mysql, it's slow

Graph data (Facebook-combined)

Graph DBMS: Neo4j

Storing Data

The whole dataset is stored as a graph in neo4j. Each point in dataset is stored as a node, and each line (an edge between two points) is stored as a relationship.

Querying Date

We use Traverser to query data. In NeighbourCount function, we set search method as breadthFirst, set relationship direction as outgoing, and set terminate evaluator as no more new nodes being added. In ReachabilityCount function, we set search method as breadthFirst, set relationship direction as outgoing, and set terminate evaluator as where last relationship type is "connects" (pruneWhereLastRelationshipTypeIs(RelTypes.CONNECTS)), which guarantees that only the nodes directly linked with startNode can be added.

Time consumption

Case/Time(ms)	1	2	3	4
Insert into DB	6482	6389	6779	6505
Count Neighbour	163	194	221	161
Count Reachablility	1907	1856	173	157

MySQL

Time consumption

Case/Time(ms)	1	2	3	4
Insert into DB	219479	210714	207847	212814
Count Neighbour	55	47	101	49
Count Reachablility	193098	188742	299	52

Comparison

CreateDB:

	minimal time	maximal time	average time
MySQL	207847	219479	213663
Neo4j	6389	7171	6447

Neighbour count:

	minimal time	maximal time	average time
MySQL	47	101	74
Neo4j	158	221	190

Reachability count :

	minimal time	maximal time	average time
MySQL	52	193098	96575
Neo4j	152	2075	1113

Summary: We can conclude that, with enough complexity of the graph dataset, the performance of CreateDB and Reachability Count on Neo4j is much better than on MySQL, and the difference between the performance of Neighbour Count on Neo4j and on MySQL is pretty small. So the whole performance of Neo4j is much better than MySQL on Graph data, especially there are huge amount of edges and nodes in graph dataset.