Introduction

This example shows a technique for aggregating time-series data over different time intervals using Mongo DB.

The data used is time series traffic sensor data. The raw data collection has 1 minute granularity. We show how to roll it up into hourly and daily collections.

Different aging policies could then be applied to each of the collections. For example, raw sensor data could be kept for 3 days, hourly sensor data kept for a week and daily data kept for a month, etc.

We also compare the performance of doing a similar aggregation task using DB2.

Mongo DB

Set Up

The script here was used to generate time-series traffic sensor data: The attached ini file was used to generate data according to the following pattern:

- 100 Sensors
- 1 reading per sensor per minute
- 1440 readings per sensor per day
- 144,000 total readings per day
- readings span 2 months, resulting in a total of 8.6 million records in the collection

Sample CSV Data is shown below:

```
LINKID,TIMESTAMP,SPEED,TIME

string,date,int,int

LINK1,2018-03-23 19:28:06,15,300

LINK2,2018-03-23 19:28:06,13,244

LINK3,2018-03-23 19:28:06,37,396

…

LINK100,2018-03-23 19:29:06,15,473

LINK1,2018-03-23 19:30:06,9,444

LINK2,2018-03-23 19:30:06,17,314

LINK3,2018-03-23 19:30:06,4,219
```

This is converted to json using the attached script. A sample JSON record is as follows:

```
{

   "LINKID":"LINK1",

   "TIMESTAMP":ISODate("2018-05-21T20:28:06Z"),

   "SPEED":30,

   "TIME":436

}
```

We create a schema for the raw collection as follows:

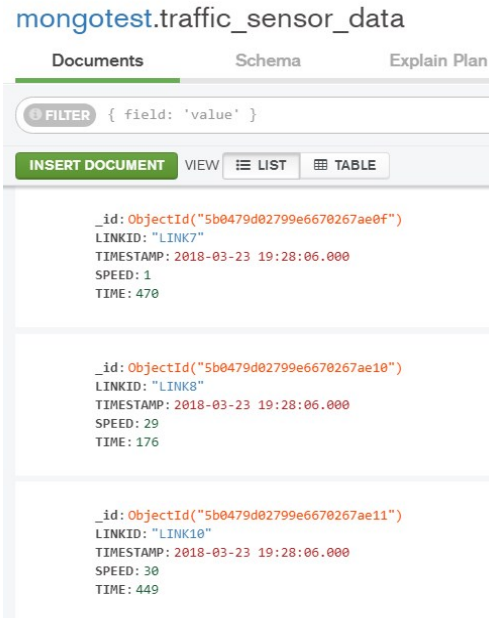
```
db.createCollection("traffic_sensor_data", {

   validator: {

      $jsonSchema: {
```

```
         bsonType: "object",
         required: [ "LINKID", "TIMESTAMP", "SPEED", "TIME" ],
         properties: {
            LINKID: {
               bsonType: "string",
               description: "must be a string and is required"
            },
            TIMESTAMP: {
               bsonType: "date",
               description: "must be a date and is required"
            },
            SPEED: {
               bsonType: "int",
               description: "must be an integer and is required"
            },
            TIME: {
               bsonType: "int",
               description: "must be an integer and is required"
            }
         }
      }
   }
})
```

A combination index is created on LINKID + TIMESTAMP, which should be a unique key:

```
db.traffic_sensor_data.createIndex( {LINKID: 1, TIMESTAMP: 1 } )
```

Mongo generates the required primary key '_id': To save space we could project LINKID+TIMESTAMP to '_id' and make it the primary key.



Hourly Roll Up

We can use an aggregation pipeline such as the one shown below to roll the sensor data up into hourly averages.

We create a derived time value 'time_hourly' which zeroes the minutes and seconds. That is then used with LINKID to group the records and calculate average speed and time values over the hour duration. More sophisticated calculations (e.g., speed variance) could also be calculated if required.. The output is placed in the collection 'traffic_sensor_data_daily' using the final $out aggregation stage:

```
db.traffic_sensor_data.aggregate([
    {
        $project: {
            _id: 0,
            LINKID: 1,
            time_hourly: { $dateFromParts: {
                                year: {$year:"$TIMESTAMP"},
                                month: {$month:"$TIMESTAMP"},
                                day: {$dayOfMonth:"$TIMESTAMP"},
                                hour: {$hour:"$TIMESTAMP"}
                            }
            },
            SPEED: 1,
            TIME: 1
        }
    },
    {
       $group: {
          _id: { LINKID: "$LINKID", TIMESTAMP: "$time_hourly" },
          mean_speed: { $avg: "$SPEED" },
          mean_time: { $avg: "$TIME" },
       }
    },
```

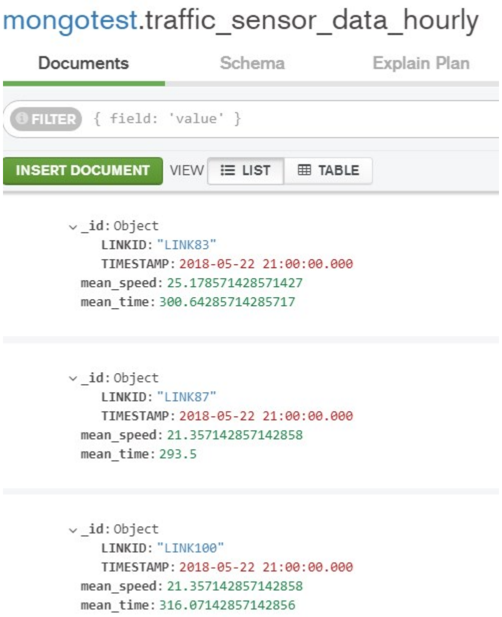
```
    { $out: "traffic_sensor_data_hourly" }

]);
```

An alternative approach to generating 'time_hourly' above would be:

```
time_hourly: { $dateFromString: {
        "dateString": { $dateToString: {
                format: "%Y-%m-%d %H:00:00.000",
                date: "$TIMESTAMP"
            }
        }
    }
},
```

However, this proves to be around 10% slower.

The hourly aggregation results in 144K hourly records:



The time for this operation on Windows was 37 seconds, so 232K records processed per second. The time on Red Hat was 46 seconds.

Scheduled Updates

A Java function to handle updates to the hourly table (e.g., scheduled to run once a day at midnight) might look something like this:

```
public static void trafficSensorDataHourly_Delta(MongoTestDriver testDriver) {

        MongoCollection<Document> trafficSensorData = testDriver.database.getCollection("traffic_sensor_data");

        AggregateIterable<Document> aggregateDocs;

        List<Document> docsAsList = new ArrayList<Document>();

        SimpleDateFormat format = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");


          try {

                Date lastCheckPoint = format.parse("2018-05-23 00:00:00");
```

```
            aggregateDocs = trafficSensorData.aggregate(Arrays.asList(
                        match(gte("TIMESTAMP",lastCheckPoint)),
                        project(fields(excludeId(),
                                    include("LINKID"),
                                    computed("time_hourly",
                                            new Document("$dateFromParts",
                                                new Document("year", new Document("$year","$TIMESTAMP"))
                                                  .append("month", new Document("$month","$TIMESTAMP"))
                                                  .append("day", new Document("$dayOfMonth","$TIMESTAMP"))
                                                  .append("hour", new Document("$hour","$TIMESTAMP"))
                                            )
                                    ),
                                    include("SPEED","TIME")
                                )
                        ),
                        group(new Document("LINKID","$LINKID").append("TIMESTAMP","$time_hourly"),
                                    avg("mean_speed", "$SPEED"),
                                    avg("mean_time", "$TIME"))
                )
            );
             for (Document doc : aggregateDocs) {
                    docsAsList.add(doc);
            }
             if (docsAsList.size() > 0)
                    testDriver.database.getCollection("traffic_sensor_data_hourly").insertMany(docsAsList);

        } catch (Exception e) {
            e.printStackTrace();
        }
}
```

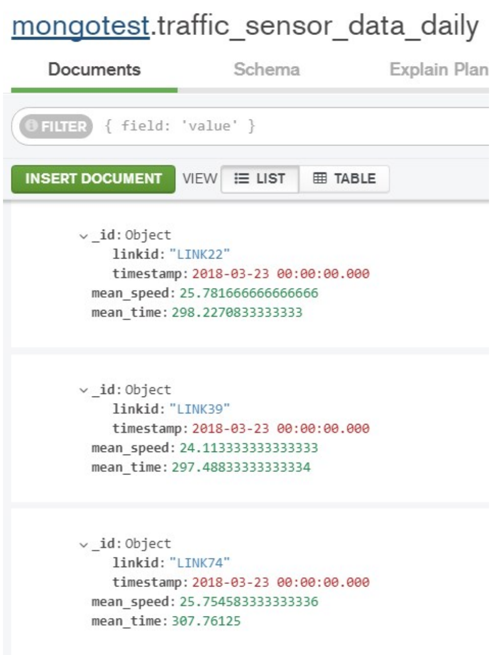
Daily Roll Up

We use a similar approach to roll up the hourly data into daily data. In this case we create a derived timestamp value 'time_daily' based on the hourly timestamp, but where we zero the hour component. The output is placed in the collection 'traffic_sensor_data_daily' using the final $out aggregation stage:

```
db.traffic_sensor_data_hourly.aggregate([
    {
        $project: {
            linkid: "$_id.LINKID",
            time_daily: { $dateFromParts: {
                                year: {$year:"$_id.TIMESTAMP"},
                                month: {$month:"$_id.TIMESTAMP"},
                                day: {$dayOfMonth:"$_id.TIMESTAMP"}
                            }
```

```
            },
            speed: "$mean_speed",
            time: "$mean_time"
        }
    },
    {
      $group: {
        _id: { linkid: "$linkid", timestamp: "$time_daily" },
        mean_speed: { $avg: "$speed" },
        mean_time: { $avg: "$time" },
      }
    },
    { $out: "traffic_sensor_data_daily" }
]);
```

This results in 6.1K records and took 775 ms to process on Windows and 800 ms on Red Hat:



Scheduled Updates

A Java function to handle updates to the daily table (e.g., scheduled to run once a week) might look something like this:

```java
public static void trafficSensorDataDaily_Delta(MongoTestDriver testDriver) {
        MongoCollection<Document> trafficSensorHourlyData = testDriver.database.getCollection("traffic_sensor_data_hourly");
        AggregateIterable<Document> aggregateDocs;
        List<Document> docsAsList = new ArrayList<Document>();
        SimpleDateFormat format = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

         try {
               Date lastCheckPoint = format.parse("2018-05-23 00:00:00");

               aggregateDocs = trafficSensorHourlyData.aggregate(Arrays.asList(
```

```
                            match(gte("_id.TIMESTAMP",lastCheckPoint)),
                            project(fields(excludeId(),
                                           computed("linkid","$_id.LINKID"),
                                           computed("timestamp",
                                                          new Document("$dateFromParts",
                                                               new Document("year", new Document("$year","$_id.TIMESTAMP"))
                                                            .append("month", new Document("$month","$_id.TIMESTAMP"))
                                                            .append("day", new Document("$dayOfMonth","$_id.TIMESTAMP"))
                                                   )
                                           ),
                                           include("mean_speed","mean_time")
                                    )
                            ),
                            group(new Document("linkid","$linkid").append("timestamp","$timestamp"),
                                           avg("mean_speed", "$mean_speed"),
                                           avg("mean_time", "$mean_time"))
                    )
            );
              for (Document doc : aggregateDocs) {
                    docsAsList.add(doc);
            }
              if (docsAsList.size() > 0)
                    testDriver.database.getCollection("traffic_sensor_data_daily").insertMany(docsAsList);
        } catch (Exception e) {
            e.printStackTrace();
        }
}
```

Map Reduce

Mongo DB also offers a native implementation of MapReduce. It doesn't perform as well as the aggregation pipeline under normal circumstances. MapReduce does offer the potential for parallelization in sharded clusters. In this case it can offer better performance at scale. For the sake of completeness we show two possible MapReduce implementations of the above hourly and daily aggregation pipelines:

Hourly Example

```
db.traffic_sensor_data.mapReduce(
  function() {
    emit( { LINKID: this.LINKID,
        TIMESTAMP: new Date( this.TIMESTAMP.getTime() - this.TIMESTAMP.getMinutes() * 60 * 1000
                            - this.TIMESTAMP.getSeconds() * 1000 )
        },
       { speed: this.SPEED, time: this.TIME }
    );
  },
  function(key, values) {
    var len = values.length;
```

```
        var speedsum = 0, timesum = 0;

        values.forEach( function(v) {
            speedsum += v.speed;
            timesum += v.time;
        });
        return { speed: speedsum/len, time: timesum/len }
    },
    {
        out : "traffic_sensor_data_hourly"
    }
)
```


```
_id: Object
    LINKID: "LINK1"
    TIMESTAMP: 2018-03-23 19:00:00.000
value: Object
    speed: 26.682638888888892
    time: 282.6125

_id: Object
    LINKID: "LINK1"
    TIMESTAMP: 2018-03-23 20:00:00.000
value: Object
    speed: 23.650282409897795
    time: 236.94988344988343
```


As predicted, performance is worse with MapReduce. See the table at the end of the page for details.

Scheduled Updates

A Java function to handle updates to the hourly table (e.g., scheduled to run once a day at midnight) might look something like this:

```
public static void trafficSensorDataHourly_delta_mapReduce(MongoTestDriver testDriver) {
        MongoCollection<Document> trafficSensorData = testDriver.database.getCollection("traffic_sensor_data");
        List<Document> docsAsList = new ArrayList<Document>();
        SimpleDateFormat format = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
        String mapFunction = "function() {\r\n" +
              "        emit( { LINKID: this.LINKID,\r\n" +
              "                TIMESTAMP: new Date( this.TIMESTAMP.getTime()\r\n" +
             "                                               - this.TIMESTAMP.getMinutes() * 60 * 1000\r\n" +
              "                                               - this.TIMESTAMP.getSeconds() * 1000 )\r\n" +
              "                },\r\n" +
              "                { speed: this.SPEED, time: this.TIME }\r\n" +
              "        );\r\n" +
              "    }";
        String reduceFunction = "function(key, values) {\r\n" +
              "        var len = values.length;\r\n" +
              "        var speedsum = 0, timesum = 0;\r\n" +
```

```
"        \r\n" +
"        for (var i=0; i<len; i++) {\r\n" +
"            speedsum += values[i].speed;\r\n" +
"            timesum += values[i].time;\r\n" +
"        });\r\n" +
"        return { speed: speedsum/len, time: timesum/len }\r\n" +
"    }";

        try {
                Date lastCheckPoint = format.parse("2018-05-23 00:00:00");
                MapReduceIterable<Document> docs = trafficSensorData.mapReduce(mapFunction,reduceFunction)
                                                                     .filter(gte("TIMESTAMP",lastCheckPoint));
                  for (Document doc : docs) {
                      docsAsList.add(doc);
                }
                 if (docsAsList.size() > 0) {
                      testDriver.database.getCollection("traffic_sensor_data_hourly").insertMany(docsAsList);
                }
        } catch (Exception e) {
                e.printStackTrace();
        }
}
```

Daily Example

```
db.traffic_sensor_data_hourly.mapReduce(
   function() {
      emit( { linkid: this._id.LINKID,
            timestamp: new Date( this._id.TIMESTAMP.getTime() - this._id.TIMESTAMP.getHours() * 60 * 60 * 1000 )
            },
           { speed: this.value.speed, time: this.value.time }
       );
   },
   function(key, values) {
      var len = values.length;
      var speedsum = 0, timesum = 0;

      values.forEach( function(v) {
          speedsum += v.speed;
         timesum += v.time;
       });
      return { speed: speedsum/len, time: timesum/len }
   },
   {
      out: "traffic_sensor_data_daily"
```

```
    }

)
```

```
_id: Object
    linkid: "LINK1"
    timestamp: 2018-03-23 00:00:00.000
value: Object
    speed: 25.870187041125813
    time: 296.4245695658898
```

```
_id: Object
    linkid: "LINK1"
    timestamp: 2018-03-24 00:00:00.000
value: Object
    speed: 24.782718938207747
    time: 307.11835971704767
```

Scheduled Updates

A Java function to handle updates to the daily table using MapReduce (e.g., scheduled to run once a week) might look something like this:

```
public static void trafficSensorDataDaily_delta_mapReduce(MongoTestDriver testDriver) {

        MongoCollection<Document> trafficSensorDataHourly = testDriver.database.getCollection("traffic_sensor_data_hourly");

        List<Document> docsAsList = new ArrayList<Document>();

        SimpleDateFormat format = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

        String mapFunction = "function() {\r\n" +

              "        emit( { linkid: this._id.LINKID,\r\n" +

              "                timestamp: new Date( this._id.TIMESTAMP.getTime() - this._id.TIMESTAMP.getHours() * 60 * 60 * 1000 )\r\n" +

              "              },\r\n" +

              "              { speed: this.value.speed, time: this.value.time }\r\n" +

              "        );\r\n" +

              "    }";

        String reduceFunction = "function(key, values) {\r\n" +
              "        var len = values.length;\r\n" +
              "        var speedsum = 0, timesum = 0;\r\n" +
              "        \r\n" +
              "        for (var i=0; i<len; i++) {\r\n" +
              "            speedsum += values[i].speed;\r\n" +
              "            timesum += values[i].time;\r\n" +
              "        });\r\n" +
              "        return { speed: speedsum/len, time: timesum/len }\r\n" +
              "    }";

          try {

                Date lastCheckPoint = format.parse("2018-05-23 00:00:00");

                MapReduceIterable<Document> docs = trafficSensorDataHourly.mapReduce(mapFunction,reduceFunction)

                                                                        .filter(gte("_id.TIMESTAMP",lastCheckPoint));

                  for (Document doc : docs) {

                        docsAsList.add(doc);

                }

                 if (docsAsList.size() > 0) {

                        testDriver.database.getCollection("traffic_sensor_data_daily").insertMany(docsAsList);

                }

        } catch (Exception e) {
```

```
            e.printStackTrace();
        }
}
```

DB2

Set Up

We load the source CSV data from the Mongo DB example directly into a DB2 table using the LOAD command:

```
create table wih.traffic_sensor_data (
  LINKID varchar(25),
  TIMESTAMP timestamp,
  SPEED INTEGER,
  TIME INTEGER
);
```

```
db2 'load from traffic_sensor_data.csv of del modified by timestampformat="YYYY-M-D HH:MM:SS" messages /tmp/sensor.log insert into wih.traffic_sensor_data'
```

```
CREATE INDEX WIH.LINK_TIMESTAMP_IDX
   ON WIH.TRAFFIC_SENSOR_DATA
   (LINKID ASC, TIMESTAMP ASC)
   CLUSTER
   MINPCTUSED 0
   ALLOW REVERSE SCANS
   PAGE SPLIT SYMMETRIC
   COLLECT SAMPLED DETAILED STATISTICS
   COMPRESS NO;
```

Hourly Roll Up

First we create the hourly table:

```
CREATE TABLE WIH.TRAFFIC_SENSOR_DATA_HOURLY (
   LINKID VARCHAR(25),
   TIMESTAMP_HOURLY TIMESTAMP,
   MEAN_SPEED INTEGER,
   MEAN_TIME INTEGER
);
```

To measure time for the operation we use a shell script as follows:

```
db2 connect to WIHDB > /dev/null

db2 truncate table wih.traffic_sensor_data_hourly > /dev/null

date1=$(date +"%s%N")

db2 -ntf insert_hourly.sql

db2 connect reset > /dev/null

date2=$(date +"%s%N")

diff=$(($date2-$date1))

echo "$(($diff/1000000000)) seconds, $(($diff % 1000000000/1000000)) milliseconds  elapsed."
```

and where 'insert_hourly.sql' is as follows:

```
INSERT INTO WIH.TRAFFIC_SENSOR_DATA_HOURLY

  SELECT LINKID, TIMESTAMP_ISO(TIMESTAMP - MINUTE(TIMESTAMP) MINUTES - SECOND(TIMESTAMP) SECONDS) as
TIMESTAMP_HOURLY,

   AVG(SPEED) AS MEAN_SPEED, AVG(TIME) AS MEAN_TIME

   FROM WIH.TRAFFIC_SENSOR_DATA

   GROUP BY LINKID, TIMESTAMP_ISO(TIMESTAMP - MINUTE(TIMESTAMP) MINUTES - SECOND(TIMESTAMP) SECONDS);
```

We're lucky in this case that the number of inserts (144K) isn't enough to saturate the transaction logs. Otherwise we would have to do the select/insert operations in batches (e.g., using a stored procedure). Perhaps surprisingly the DB2 aggregation is faster than it was for Mongo at 22 seconds. That may be because we have a cluster index on the LINKID+TIMESTAMP. The result is as follows:

WIH.TRAFFIC_SENSOR_DATA_HOURLY

LINKID [VARCHAR(25)]	TIMESTAMP_HOURLY [TIMESTAMP]	MEAN_SPEED [INTEGER]	MEAN_TIME [INTEGER]
LINK1	2018-03-23 19:00:00.0	24	299
LINK10	2018-03-23 19:00:00.0	25	268
LINK100	2018-03-23 19:00:00.0	23	267
LINK11	2018-03-23 19:00:00.0	26	314
LINK12	2018-03-23 19:00:00.0	25	313
LINK13	2018-03-23 19:00:00.0	27	287
LINK14	2018-03-23 19:00:00.0	25	294
LINK15	2018-03-23 19:00:00.0	27	321
LINK16	2018-03-23 19:00:00.0	27	284
LINK17	2018-03-23 19:00:00.0	28	267
LINK18	2018-03-23 19:00:00.0	26	315
LINK19	2018-03-23 19:00:00.0	21	302

Daily Roll Up

Before doing the daily calculation we create a cluster index on the hourly table:

```
CREATE INDEX WIH.LINK_TIMESTAMP_HOURLY_IDX

   ON WIH.TRAFFIC_SENSOR_DATA_HOURLY

   (LINKID ASC, TIMESTAMP_HOURLY ASC)

   CLUSTER

   MINPCTUSED 0

   ALLOW REVERSE SCANS
```

```
   PAGE SPLIT SYMMETRIC
   COLLECT SAMPLED DETAILED STATISTICS
   COMPRESS NO;
```

Next we create the daily table:

```
CREATE TABLE WIH.TRAFFIC_SENSOR_DATA_DAILY (
   LINKID VARCHAR(25),
   TIMESTAMP_DAILY TIMESTAMP,
   MEAN_SPEED INTEGER,
   MEAN_TIME INTEGER
);
```

Again, we use a shell script to invoke the operation and measure the duration:

```
db2 connect to WIHDB > /dev/null
db2 truncate table wih.traffic_sensor_data_daily > /dev/null

date1=$(date +"%s%N")
db2 -ntf insert_daily.sql
db2 connect reset > /dev/null
date2=$(date +"%s%N")
diff=$(($date2-$date1))
echo "$(($diff/1000000000)) seconds, $(($diff % 1000000000/1000000)) milliseconds  elapsed."
```

and where 'insert_daily.sql' is as follows:

```
INSERT INTO WIH.TRAFFIC_SENSOR_DATA_DAILY
  SELECT LINKID, TIMESTAMP_ISO(TIMESTAMP_HOURLY - HOUR(TIMESTAMP_HOURLY) HOURS) as TIMESTAMP_DAILY,
   AVG(MEAN_SPEED) AS MEAN_SPEED, AVG(MEAN_TIME) AS MEAN_TIME
   FROM WIH.TRAFFIC_SENSOR_DATA_HOURLY
   GROUP BY LINKID, TIMESTAMP_ISO(TIMESTAMP_HOURLY - HOUR(TIMESTAMP_HOURLY) HOURS);
```

The time for the operation was 411 ms, again quicker than Mongo DB. The output looks as follows:

WIH.TRAFFIC_SENSOR_DATA_DAILY

LINKID [VARCHAR(25)]	TIMESTAMP_DAILY [TIMESTAMP]	MEAN_SPEED [INTEGER]	MEAN_TIME [INTEGER]
LINK1	2018-03-23 00:00:00.0	25	304
LINK10	2018-03-23 00:00:00.0	25	300
LINK100	2018-03-23 00:00:00.0	25	296
LINK11	2018-03-23 00:00:00.0	24	299
LINK12	2018-03-23 00:00:00.0	24	311
LINK13	2018-03-23 00:00:00.0	25	298
LINK14	2018-03-23 00:00:00.0	25	297
LINK15	2018-03-23 00:00:00.0	24	311
LINK16	2018-03-23 00:00:00.0	24	297
LINK17	2018-03-23 00:00:00.0	27	291
LINK18	2018-03-23 00:00:00.0	26	308

Performance Comparison between DB2 and Mongo DB

	Total Raw Records	Hourly Aggregation		Daily Aggregation	
		Aggregate Record Count	Aggregation Time (ms)	Aggregate Record Count	Aggregation Time (ms)
DB2	8.64 million	144,100	22,000	1441	411
Mongo DB					
Aggregation Pipeline	8.64 million	144,100	37,000	1441	775
MapReduce			144,000		2500

So, for this particular use case DB2 performs better. However, bear in mind that Mongo DB's aggregation pipeline feature allows for many more complex operations than DB2 offers, plus its syntax is a lot easier to formulate and understand.