Environments

- Mongo Community was installed on a laptop with Windows 7. Installation used the instructions here.
- Mongo Enterprise Edition was installed on Red Hat Linux 7. Installation used the attached script (there were issues with standard install methods).
- IOC was installed on four Red Hat 6 servers.

Operating System	Database	CPU Cores	RAM	DISK
Windows 7, 64-bit	Mongo Community Edition 3.4	8	16 GB	SSHD (hybrid SSD drives)
Red Hat Linux Server 7.4 (Maipo)	Mongo Enterprise Server Edition 3.4	8	64 GB	SAS 10K RPM
Red Hat Linux Server 6.9 (Santiago)	IOC (DB2 10.5)	8	64 GB	SAS 10K RPM

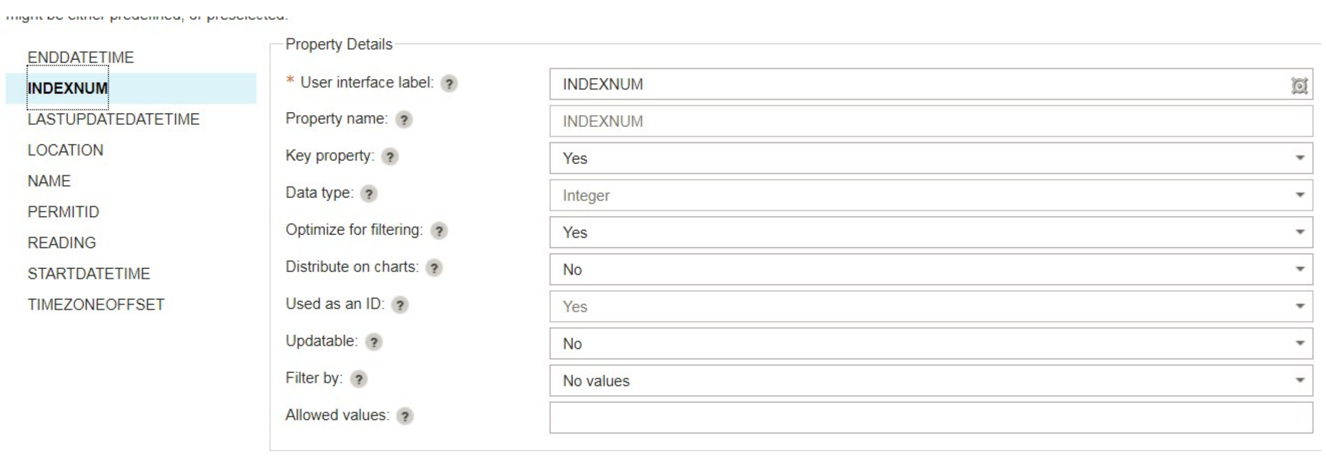
Data Source Data

IOC

The script here was used to generate 1.7 million rows of data - 42K records per day over 40 consecutive days. For example:

```
INDEXNUM,STARTDATETIME,ENDDATETIME,LOCATION,NAME,LASTUPDATEDATETIME,TIMEZONEOFFSET,PERMITID,READING
1,2017-12-01 00:00:00.000,2017-12-10 00:00:00.000,POINT(4.983074 47.666847),Large Wildfire Moving Rapidly West,  2017-12-20 00:00:00.000,0,666122, 80.276777
2,2017-12-01 00:00:02.000,2017-12-10 00:00:02.000,POINT(0.370679 49.231539),Hurricane Approaching,              2017-12-20 00:00:01.000,0,2516938,40.260140
3,2017-12-01 00:00:04.000,2017-12-10 00:00:04.000,POINT(3.328866 49.582488),Multi-vehicle Accident with Injuries,2017-12-20 00:00:02.000,0,922566, 97.003961
```

The column 'INDEXNUM' was configured as a 'used as id' column of type INTEGER. Thus, IOC maintains a unique index on the column. Indexes are also created by IOC on start/end/lastupdate dates. A spatial index is created on the LOCATION column. The events span an area around Central France.



might be either predefined, or preselected.

ENDDATETIME
INDEXNUM
LASTUPDATEDATETIME
LOCATION
NAME
PERMITID
READING
STARTDATETIME
TIMEZONEOFFSET

Property Details

* User interface label:	INDEXNUM
Property name:	INDEXNUM
Key property:	Yes
Data type:	Integer
Optimize for filtering:	Yes
Distribute on charts:	No
Used as an ID:	Yes
Updatable:	No
Filter by:	No values
Allowed values:	

Mongo DB

The attached python script was used to convert from CSV to JSON for import into MongoDB. There are probably open-source tools available to do the same thing.

Below is an example of one of the JSON records (where dates are of type ISODate()):

```
{
 "indexnum":1,
 "startdatetime":ISODate("2017-12-01T00:00:00.000Z"),
 "enddatetime":ISODate("2017-12-10T00:00:00.000Z"),
 "location":{
   "coordinates":[
       4.983074,
       47.666847
     ],
   "type":"Point"
 },
 "name":"Large Wildfire Moving Rapidly West",
 "lastupdatedatetime":ISODate("2017-12-20T00:00:00.000Z"),
 "timezoneoffset":0,
 "permitid":666122,
 "reading":80.276777
}
```

Although schemas in Mongo are flexible and can vary from one document in a collection to another, we decided to impose a schema on the collection to be as consistent with the IOC data source as possible:

```
db.createCollection("weather_events", {
  validator: {
    $jsonSchema: {
      bsonType: "object",
      required: [ "indexnum", "startdatetime", "enddatetime", "location", "name", "lastupdatedatetime", "timezoneoffset", "permitid","reading" ],
      properties: {
        indexnum: {
          bsonType: "int",
           description: "must be an integer and is required"
         },
        startdatetime: {
          bsonType: "date",
           description: "must be a date and is required"
         },
        enddatetime: {
          bsonType: "date",
           description: "must be a date and is required"
         },
        name: {
          bsonType: "string",
           description: "must be a string and is required"
         },
        lastupdatedatetime: {
          bsonType: "date",
            description: "must be a date and is required"
          },
          timezoneoffset: {
            bsonType: "int",
            description: "must be an integer and is required"
          },
          permitid: {
            bsonType: "int",
            description: "must be an integer and is required"
          },
          reading: {
            bsonType: "double",
            description: "must be a double and is required"
          }
        }
      }
    }
})
```

The data was imported into the weather_events collection (in the Mongotest database) using the mongoimport tool:

```
mongoimport --file weather_events.json --collection weather_events --db mongotest --type json --mode
insert
```

A spatial index was created on the LOCATION column as follows:

```
db.weather_events.createIndex( {location:"2dsphere"} )
```

Indexes were also created on the DATE and INDEXNUM columns:

```
db.weather_events.createIndex( {startdatetime: 1 } )
db.weather_events.createIndex( {enddatetime: 1 } )
db.weather_events.createIndex( {lastupdatedatetime: 1 } )
db.weather_events.createIndex( {indexnum: 1 } )
```

Ingestion Rates

	Number of Columns/Fields	Ingestion Rate
IOC No 'Used as an ID' column configured	31*	2100 per second

'Used as an ID' column configured		1484 per second
DB2 (Load command)	9	> 100,000 per second*
MONGO DB Windows Linux	9	 42,000 per second 42,000 per second

* IOC adds several generated columns (e.g., STARTYEAR, STARTHOUR, ENDYEAR, etc.).

** The performance of DB2's load command will vary according to the number and type of columns. See here for some examples.

Storage Usage

IOC Queries (kB):

```
select tabname,data_object_p_size from sysibmadm.admintabinfo where tabschema='IOC' and tabname = 'TARGET_TABLE_WEATHER_EVENTS'

select tabname,index_object_p_size from table(admin_get_index_info('','IOC','TARGET_TABLE_WEATHER_EVENTS')) group by tabname,index_object_p_size
```

Mongo DB Queries (bytes):

```
> db.weather_events.stats().storageSize

134197248

> db.weather_events.stats().totalIndexSize

100573184

> db.weather_events.stats().wiredTiger.cache

...

    "bytes currently in the cache" : 520586150,

...
```

	Records	Table Storage (MB)	Index Storage (MB)	Total Storage (MB)	Memory Used (MB)
IOC	1,692,000	225	246	471	
Mongo DB	1,692,000	134	100*	234	520

* In this particular case we could have specified the 'indexnum' column as the primary key column (instead of having MongoDB autogenerate an '_id' column). That would have saved a further 16 MB of storage and a corresponding amount of RAM.

Data Aging in Mongo DB

Mongo implements data aging using TTL (time-to-live) indexes (link).

For example, to age documents where the field LASTUPDATEDATETIME is older than 30 days:

```
db.weather_events.createIndex( { "lastupdatedatetime": 1 }, { expireAfterSeconds: 2592000 } )
```

Each document in a collection could also be expired at a specific date by setting 'expireAfterSeconds' to 0. The date field will then hold the time when the document should be expired by Mongo DB:

```
db.weather_events.createIndex( { "expirationDate": 1 }, { expireAfterSeconds: 0 } )
```

Using the aggregation pipeline, finer resolution data could be summarized into daily/monthly/yearly rollup collections before it has been aged.

Encryption

Since version 3.2 the default storage engine used by Mongo DB, WiredTiger, provides encryption at rest. More details here.

Other security features are described here.

Basic Tests

Standard Indexed Queries

IOC

We executed two count queries against the IOC data source, one on startdatetime and one on indexnum (both columns indexed):

```
select count from ioc.target_table_weather_events where startdatetime > '2017-12-23 00:00:00' and startdatetime < '2017-12-25 00:00:00';

select count from ioc.target_table_weather_events where indexnum > 10 and indexnum < 20;
```

To capture timing we created a simple shell script as follows:

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

echo "Date Query"

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

db2 "select count from ioc.target_table_weather_events where startdatetime > '2017-12-23 00:00:00' and startdatetime < '2017-12-25 00:00:00'";

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

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

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

echo "Numerical query"

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

db2 "select count from ioc.target_table_weather_events where indexnum > 10 and indexnum < 20";

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

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

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

db2 connect reset > /dev/null
```

Mongo DB

From the Mongo Shell the two equivalent queries are as follows:

```
db.weather_events.find( { startdatetime: {$gt: ISODate("2017-12-23 00:00:00"), $lt: ISODate("2017-12-25 00:00:00")} } ).count()

db.weather_events.find( { indexnum: {$gt: 10, $lt: 20} }).count()
```

To capture timing in the Mongo case a simple Java program was created. There are three basic ways to formulate queries like the above examples using the Mongo Java driver:

1. Use Document.parse() to parse the Json string:

```
   String dateQuery = "{ startdatetime: {$gt: ISODate(\"2017-12-23 00:00:00\"), $lt: ISODate(\"2017-12-25 00:00:00\")} }"

   long count = weatherEvents.count(Document.parse(dateQuery));
```

2. Use the 'new Document()' pattern:

```
    count = weatherEvents.count(new Document("indexnum",new Document("$gt",10).append("$lt", 20)))
```

3. Use helper functions com.mongodb.client.model.Filters.* class:

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

count = weatherEvents.count(and(gt("startdatetime", format.parse("2017-12-23 00:00:00")),

                                             lt("startdatetime", format.parse("2017-12-25 00:00:00"))));

count = weatherEvents.count(and(gt("indexnum", 10), lt("indexnum", 20)));
```

For a basic count query the times for each of the three methods were as follows:

- Method 1: 8.4 ms
- Method 2: 3 ms
- Method 3: 1 ms

On that basis Method 3 was used for the benchmark timings.

Query	IOC	MongoDB
Date	86 ms	73 ms
Count	9 ms	1 ms

Geospatial Queries

We measured response times for a count and a paged query over a range of four zoom levels. These are typical queries used by IOC's data source framework.

IOC

```
SELECT assessmentdatetime,

     startdatetime,

     objectid,

     deleteflag,

     lastupdatedatetime,

     indexnum,

     enddatetime,

     classification,

     annotationid,

     timezoneoffset,

     name,

     location ..st_astext AS Location

FROM   ioc.target_view_weather_events

WHERE  db2gse.St_intersects(location, db2gse.St_buffer

(db2gse.St_geometry(CAST('POLYGON((2.551103 45.881680,

3.056474 45.881680,

3.056474 46.037291,

2.551103 46.037291,

2.551103 45.881680))' AS CLOB),1003),

 0)) = 1 selectivity 0.000001

FETCH first 5001 ROWS only
```

```
OPTIMIZE FOR 5001 ROWS;

select count from (SELECT assessmentdatetime,
      startdatetime,
      objectid,
      deleteflag,
      lastupdatedatetime,
      indexnum,
      enddatetime,
      classification,
      annotationid,
      timezoneoffset,
      name,
      location ..st_astext AS Location
FROM   ioc.target_view_weather_events
WHERE  db2gse.St_intersects(location, db2gse.St_buffer
(db2gse.St_geometry(CAST('POLYGON((2.551103 45.881680,
3.056474 45.881680,
3.056474 46.037291,
2.551103 46.037291,
2.551103 45.881680))' AS CLOB),1003),
 0)) = 1 selectivity 0.000001);
```

The coordinates above are varied according to zoom level. The values above are for the maximum zoom level tested.

Mongo DB

The equivalent queries in the Mongo Shell would be as follows (again, coordinates will vary according to zoom level):

```
db.weather_events.find( { location: { $geoIntersects: { $geometry: { type: "Polygon" , coordinates:  [ [ [ 2.551103, 45.881680 ], [ 3.056474, 45.881680 ], [ 3.056474, 46.037291 ], [ 2.551103, 46.037291 ], [ 2.551103, 45.881680 ] ] ] } } } } ).count()
```

```
db.runCommand( { find: "weather_events", filter: { location: { $geoIntersects: { $geometry: { type: "Polygon" , coordinates: [ [ [ 2.551103, 45.881680 ], [ 3.056474, 45.881680 ], [ 3.056474, 46.037291 ], [ 2.551103, 46.037291 ], [ 2.551103, 45.881680 ] ] ] } } } }, limit: 5000, batchSize: 5000 } )
```

For timing purposes the queries were executed using a Java program.

Performance Comparison

The number of matching items (out of a total of 1.68 million) at each of the four zoom levels tested is as follows:

- Zoom 1: 847,876 (50%)
- Zoom 2: 284,676 (17%)
- Zoom 3:   70,912 (4%)
- Zoom 4:   4532 (0.02%)

Query	Zoom Level	IOC	Mongo (Windows)	Mongo (Linux)
	1	32,321 ms	6912 ms	12,629 ms

Count	2	11,086 ms	2808 ms	5330 ms
	3	2914 ms	683 ms	1288 ms
	4	314 ms	48 ms	90 ms
Paging	1	2599 ms	129 ms	228 ms
	2	973 ms	101 ms	150 ms
	3	550 ms	103 ms	136 ms
	4	374 ms	75 ms	121 ms

Mongo performs significantly better than IOC in all cases. Interestingly, Mongo Community Edition on Windows performs much better than Mongo Enterprise Edition on Linux Red Hat 7. That goes against the conventional wisdom of online discussions. At this point it's not clear why Windows performance in this case is better. It may be down to the hybrid SSH drives used on the Windows machine.