Using Pluggable Apache Spark SQL Filters to help GridPocket users keep up with the Jones' (and save the planet)

Paula Ta-Shma,

Guy Gerson

IBM Research

Contact: paula@il.ibm.com

Joint work with:
Tal Ariel, IBM
Filip Gluszak, GridPocket
Gal Lushi, IBM
Yosef Moatti, IBM
Papa Niamadio, GridPocket
Nathaël Noguès, GridPocket

You have 5 minutes to pack your bags…

?

Using Pluggable Apache Spark SQL Filters to help GridPocket users keep up with the Jones' (and save the planet)

Paula Ta-Shma,

Guy Gerson

IBM Research

Contact: paula@il.ibm.com

Joint work with:
Tal Ariel, IBM
Filip Gluszak, GridPocket
Gal Lushi, IBM
Yosef Moatti, IBM
Papa Niamadio, GridPocket
Nathaël Noguès, GridPocket

What would convince you to reduce your energy usage ?

What would convince you to reduce your energy usage ?

- **Save $?**

What would convince you to reduce your energy usage ?

- **Save $?**
- **Save the planet ?**

Competing to Save Energy

- **Research shows that utility customers are most influenced by peer pressure to save energy**
- **=> Help utilities compare their customers' energy consumption with that of their neighbours**
  - Queries are anonymized

GridPocket

- A smart grid company developing energy management applications and cloud services for electricity, water and gas utilities
- HQ based in France
- http://www.gridpocket.com/
- Developed open source data generator
- Provided industry specific use cases and algorithms

The GridPocket Dataset

- Our target: 1 million meters reporting every 15 minutes
- Records are ~100 bytes
- Generated ~1TB in 3 months
- Allowing for 1 order of magnitude growth in each dimension gives 1PB in 3 months
- => Use object storage
- Typically at least order of magnitude lower storage cost than NoSQL databases
- NoSQL dataset – one large denormalized table containing meter reading information
  - SQL query for nearest neighbours

What is Object Storage ?

- **Objects contain data and metadata**
- **Written once and not modified**
  - Cannot append or update data
    - Can overwrite completely
    - Can update metadata
  - No rename operation
- **Accessed through RESTful HTTP**
  - PUT/GET/POST/DELETE object/bucket
  - Flat namespace
- **High capacity, low cost**
- **Storage of choice for Big datasets**
  - Analytics works best on equally sized objects
- **Examples**
  - Amazon S3, IBM COS, Google Cloud Storage, OpenStack Swift

Object Storage and Spark are separately managed microservices

Object Storage and Spark are separately managed microservices

Want to query data on object storage directly

Object Storage and Spark are separately managed microservices

Want to query data on object storage directly

APACHE Spark™

Object Storage and Spark are separately managed microservices

Want to query data on object storage directly

APACHE Spark™

Our goal:Minimize

1. **Number of bytes shipped**
2. **Number of REST requests**

THE key factors affecting cost (and performance)

How is this done today ?

1. **Use specialized column based formats such as Parquet, ORC**
   - Column wise compression
   - Column pruning
   - Specialized metadata

How is this done today ?

1. **Use specialized column based formats such as Parquet, ORC**
   - Column wise compression
   - Column pruning
   - Specialized metadata

We want one solution for all data formats

How is this done today ?

1. **Use specialized column based formats such as Parquet, ORC**
   - Column wise compression
   - Column pruning
   - Specialized metadata

We want one solution for all data formats

2. **Use Hive Style Partitioning to layout Object Storage data**
   - Partition pruning

Hive Style Partitioning and Partition Pruning

- **Relies on a dataset naming convention**
- **Object storage has flat namespace, but can create a virtual folder hierarchy**
  - Use '/' in object name
- **Data can be partitioned according to a column e.g. dt**
  - Information about object contents is encoded in object name e.g. **dt=2015-09-14**
- **Spark SQL can query fields in object name as well as in data e.g. "dt"**
  - Filters the objects which need to be read from Object Storage and sent to Spark

```
GPMeterStream/dt=2017-08-21/4-I-80000-120000.csv
GPMeterStream/dt=2017-08-21/4-I-800000-840000.csv
GPMeterStream/dt=2017-08-21/4-I-840000-880000.csv
GPMeterStream/dt=2017-08-21/4-I-880000-920000.csv
GPMeterStream/dt=2017-08-21/4-I-920000-960000.csv
GPMeterStream/dt=2017-08-21/4-I-960000-1000000.csv
GPMeterStream/dt=2017-08-22/1-I-0-40000.csv
GPMeterStream/dt=2017-08-22/1-I-120000-160000.csv
GPMeterStream/dt=2017-08-22/1-I-160000-200000.csv
GPMeterStream/dt=2017-08-22/1-I-200000-240000.csv
GPMeterStream/dt=2017-08-22/1-I-240000-280000.csv
```

Limitations of Today's Hive Style Partitioning in Spark

- **Only one hierarchy is possible**
  - Like database primary key, no secondary keys
- **Changing partitioning scheme requires rewriting entire dataset !**
  - Hierarchy cannot be changed without renaming all objects
- **No range partitioning**
  - Only supports partitioning with discrete types e.g. Gender:M/F, date, age etc.
  - doesn't work well for timestamps, arbitrary floating point numbers etc.
- **A deep hierarchy may result in small and non uniform object sizes**
  - May reduce performance

Can we do more ?

Can we do more ?

- **Generate metadata per object column and index it**
- **Various index types**
  - Min/max, bounding boxes, value lists
- **Filter objects according to this metadata**
- **Applies to all formats e.g. json, csv**

```
{
   "name": "GPMeterStream/dt=2017-08-17/part-00088.csv",
   "metadata": {
     "location": [
       {
         "lat": 47.5,
         "lon": 4.2
       },
        ...,
       {
         "lat": 47.6,
         "lon": 3.4
       }
     ],
     "city": {
       "set": [
         "Kilstett",
        ...,
         "Haussignémont"
       ]
     },
     "temp": {
       "min": 7.97,
       "max": 26.77
     }
   }
}
```

Can we do more ?

- **Generate metadata per object column and index it**
- **Various index types**
  - Min/max, bounding boxes, value lists
- **Filter objects according to this metadata**
- **Applies to all formats e.g. json, csv**

```
{
   "name": "GPMeterStream/dt=2017-08-17/part-00088.csv",
   "metadata": {
     "location": [
       {
         "lat": 47.5,
         "lon": 4.2
       },
        ...,
       {
         "lat": 47.6,
         "lon": 3.4
       }
     ],
     "city": {
       "set": [
         "Kilstett",
        ...,
         "Haussignémont"
       ]
     },
     "temp": {
       "min": 7.97,
       "max": 26.77
     }
   }
}
```

Min/max values

Can we do more ?

- **Generate metadata per object column and index it**
- **Various index types**
  - Min/max, bounding boxes, value lists
- **Filter objects according to this metadata**
- **Applies to all formats e.g. json, csv**

```
{
  "name": "GPMeterStream/dt=2017-08-17/part-00088.csv",
  "metadata": {
    "location": [
      {
        "lat": 47.5,
        "lon": 4.2
      },
      ...,
      {
        "lat": 47.6,
        "lon": 3.4
      }
    ],
    "city": {
      "set": [
        "Kilstett",
        ...,
        "Haussignémont"
      ]
    },
    "temp": {
      "min": 7.97,
      "max": 26.77
    }
  }
}
```

One or more bounding boxes

Can we do more ?

- **Generate metadata per object column and index it**
- **Various index types**
  - Min/max, bounding boxes, value lists
- **Filter objects according to this metadata**
- **Applies to all formats e.g. json, csv**

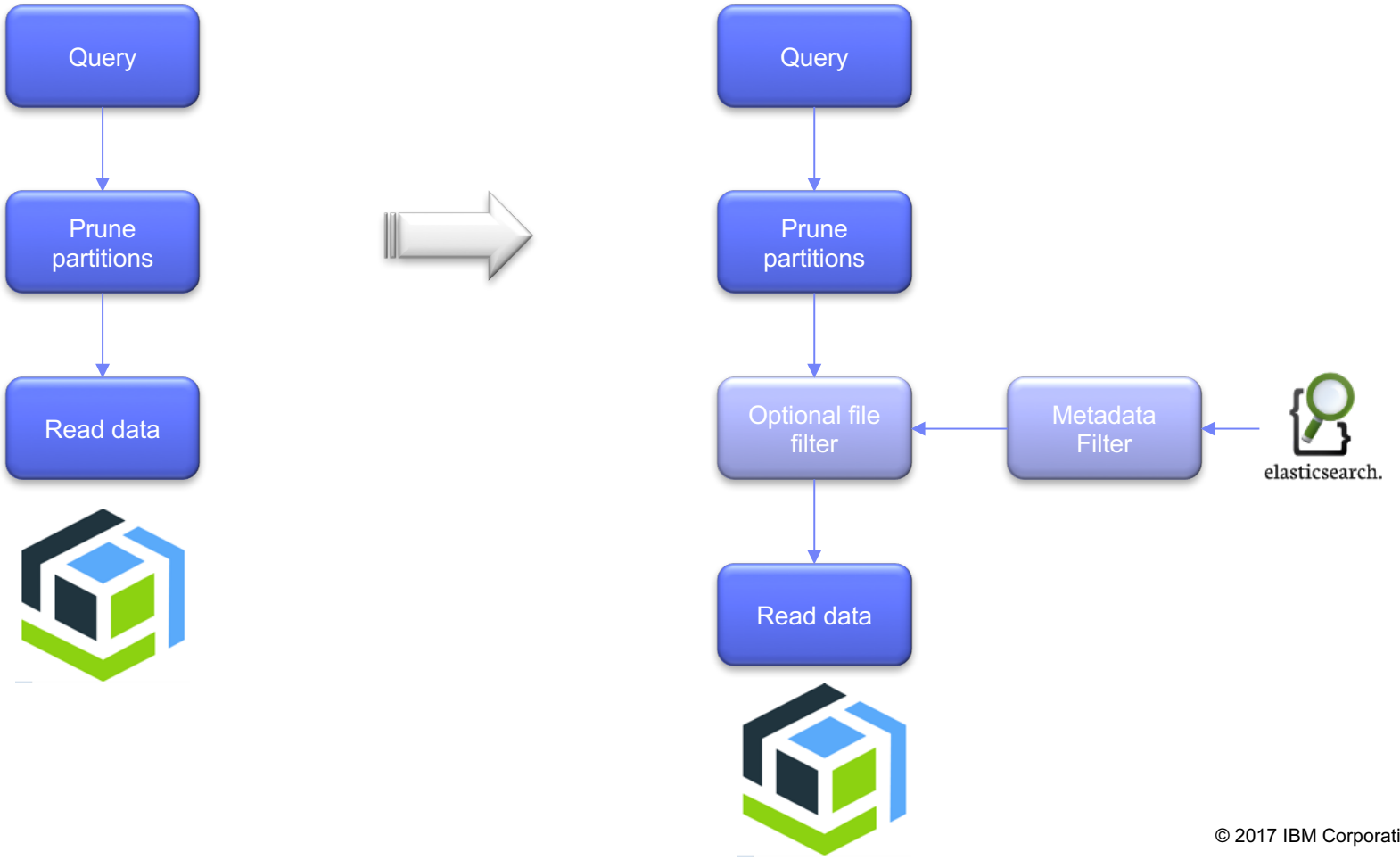
```
{
   "name": "GPMeterStream/dt=2017-08-17/part-00088.csv",
   "metadata": {
     "location": [
       {
         "lat": 47.5,
         "lon": 4.2
       },
        ...,
       {
         "lat": 47.6,
         "lon": 3.4
       }
     ],
     "city": {
       "set": [
         "Kilstett",
        ...,
         "Haussignémont"
       ]
     },
     "temp": {
       "min": 7.97,
       "max": 26.77
     }
   }
}
```

A set of values

Filter According to Metadata

- **Store one metadata record per object**
  - Unlike database fully inverted index
- **Various index types**
  - Min and max value for ordered columns
  - Bounding boxes for geospatial data
  - Bloom filters as space efficient value lists
- **Users can choose which columns to index and index type per column**
  - Can index additional columns later on
- **Main requirement: no false negatives**
- **Avoids touching irrelevant objects altogether**
- **Handles updates (PUT/DELETE object)**
  - Filtering out irrelevant objects always works

Spark SQL query execution flow



The Interface

Filters should extend this trait and implement application specific filtering logic.

```
trait ExecutionFileFilter {
 /**
   * @param dataFilters query predicates for data columns
   * @param f represents an object which exists in the file catalog
   * @return true if the object needs to be scanned during execution
   */
 def isRequired(dataFilters: Seq[Filter], f: FileStatus) : Boolean
}
```

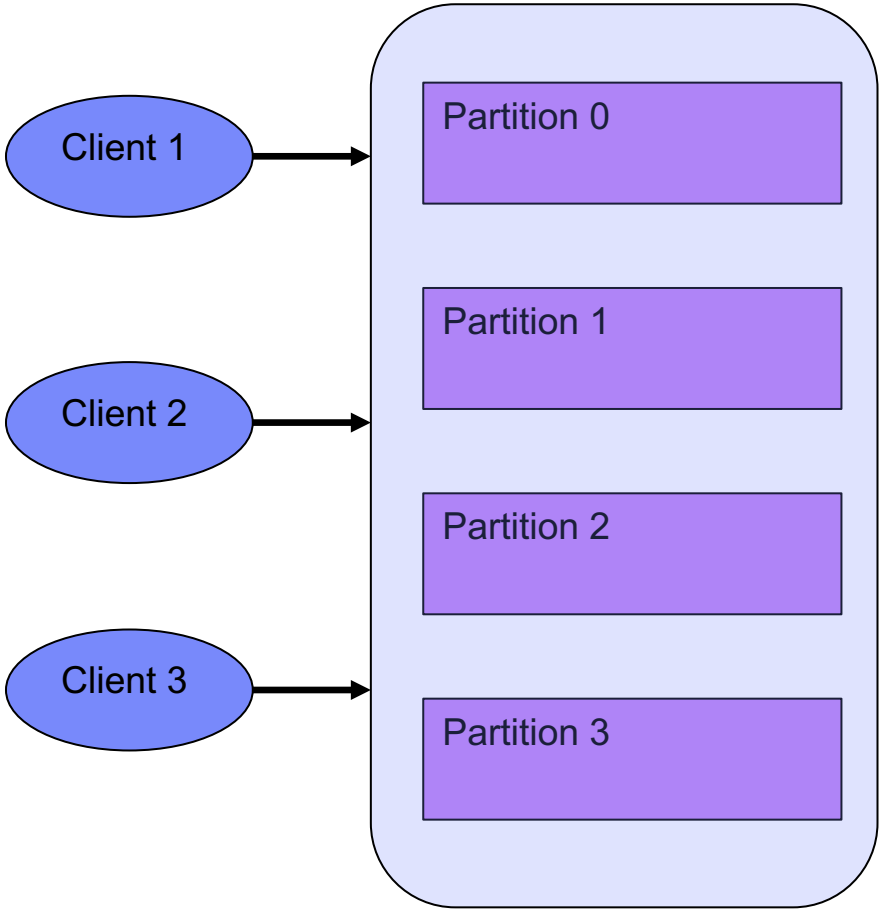
Turn filter on by specifying Filter class:

```
sqlContext.setConf("spark.sql.execution.fileFilter", “TestFileFilter”)
```

< 20 lines of code to integrate into Spark

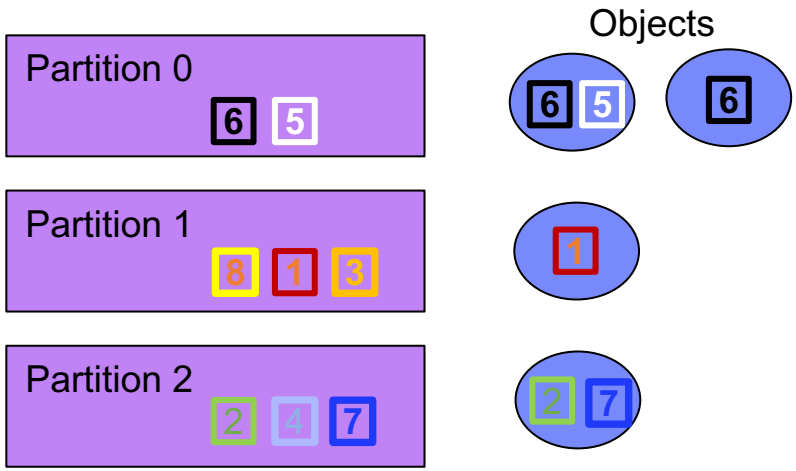
Data Ingest: How to best Partition the Data

- **Organizing data in objects in a smart way generates more effective metadata**
- **Partition during data ingestion**
- **Example: Kafka allows user to define partitioning function**
- **Needs to scale horizontally – want stateless partitioner**



Geospatial Partitioning: Grid Partitioner

- **Divide the world map into a grid**
  - Precision depends on use case
- **Each data point belongs to a cell**
- **Each cell is hashed to a partition**
- **A partition can contain multiple cells**
- **Partitions periodically generate objects**
- **Each object is described using a list of bounding boxes**
  - 1 per participating cell



Experimental Results

The GridPocket Dataset

- 1 million meters reporting every 15 minutes
- Records are ~100 bytes
- Generated ~1TB in 3 months
- Partitioned using Grid Partitioner using precision 1 (cells are roughly 10 km^2)
- Compared using 50 and 100 Kakfa partitions

Get my neighbours' average usage

```
SELECT AVG(usage)
FROM (
  SELECT vid as meter_id,
              (MAX(index)-MIN(index)) as usage
  FROM dataset
  WHERE (lat BETWEEN 43.300 AND 44.100)
  AND (lng BETWEEN 6.800 AND 7.600)
  GROUP BY vid
)
```

Experimental Results

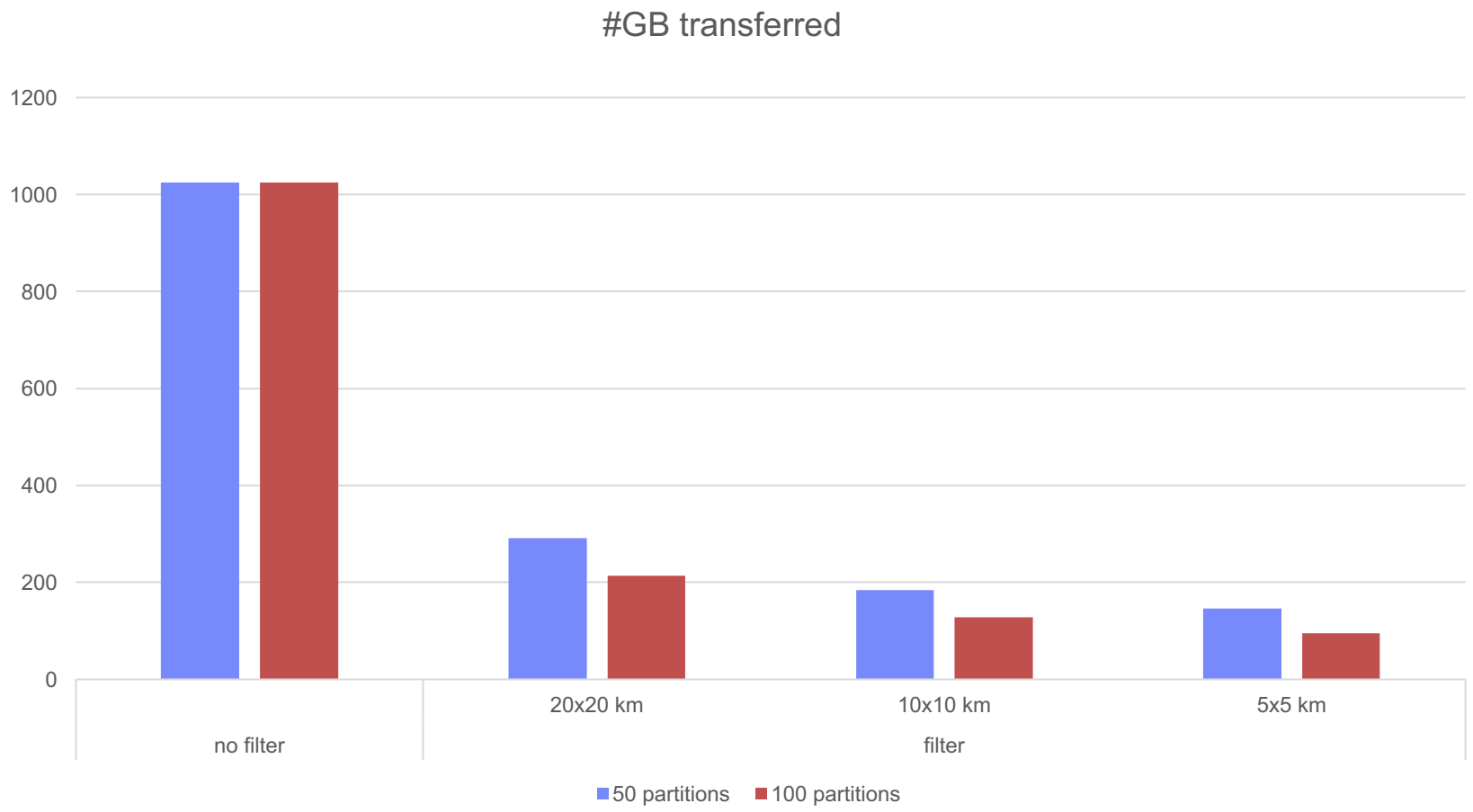
The GridPocket Dataset

- 1 million meters reporting every 15 minutes
- Records are ~100 bytes
- Generated ~1TB in 3 months
- Partitioned using Grid Partitioner using precision 1 (cells are roughly 10 km^2)
- Compared using 50 and 100 Kakfa partitions

Get my neighbours' average usage

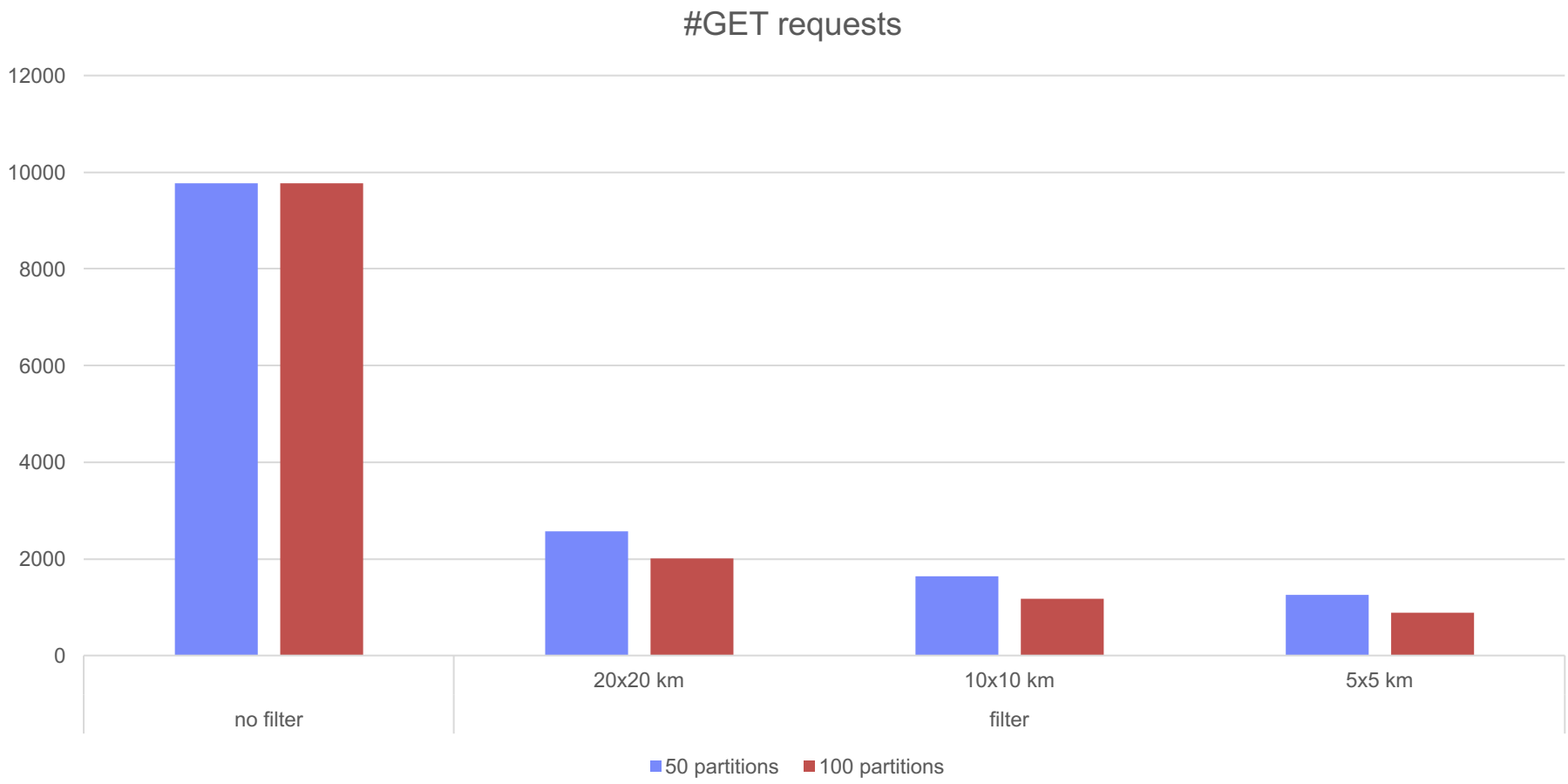
```
SELECT AVG(usage)
FROM (
  SELECT vid as meter_id,
              (MAX(index)-MIN(index)) as usage
  FROM dataset
  WHERE (lat BETWEEN 43.300 AND 44.100)
  AND (lng BETWEEN 6.800 AND 7.600)
  GROUP BY vid
)
```

#GB Transferred for bounding box queries



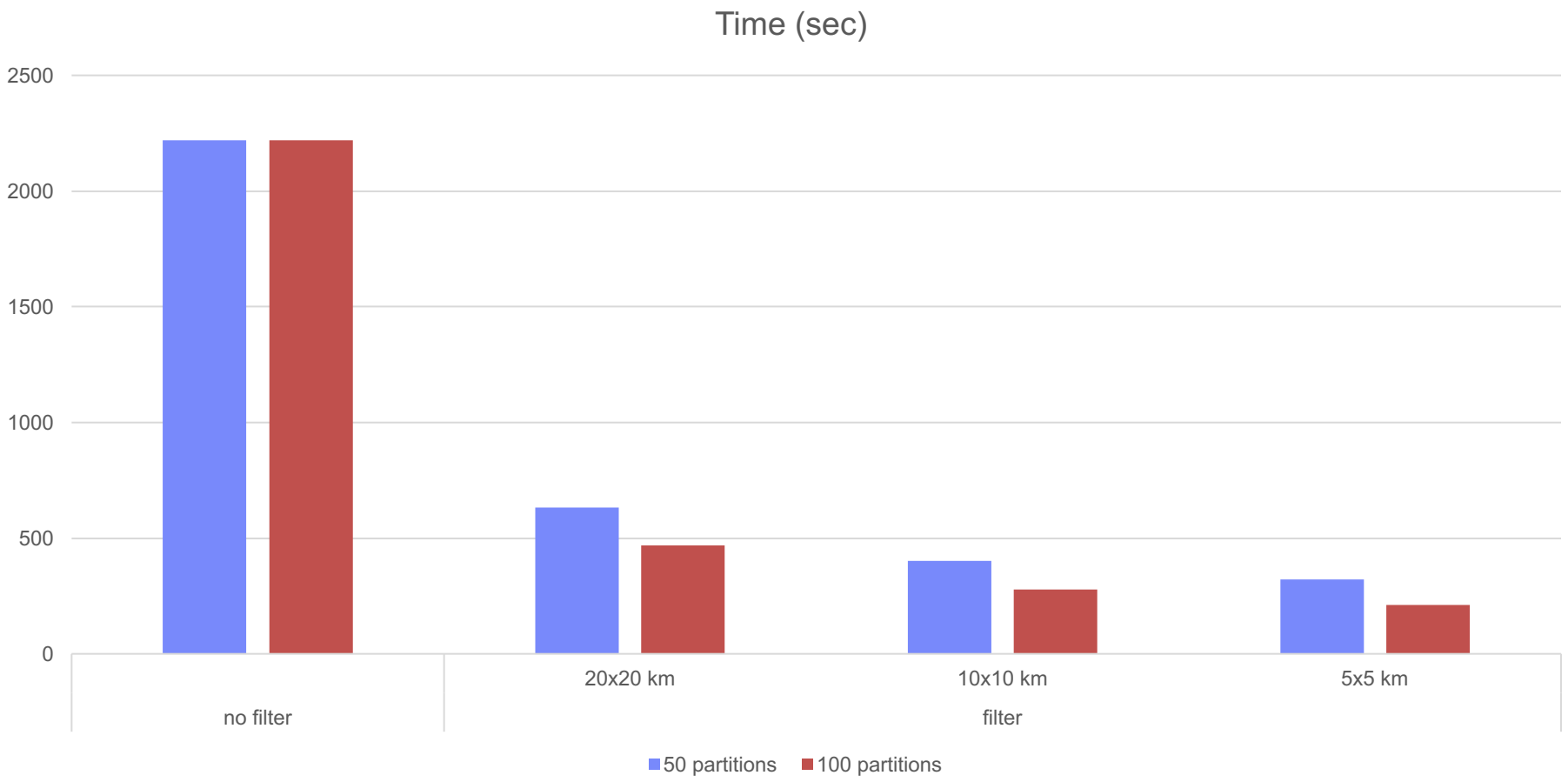
1 TB dataset, grid partitioner, precision 1, average of 10 randomly located queries

#GET Requests for bounding box queries



1 TB dataset, grid partitioner, precision 1, took average of 10 randomly located queries

Time (sec) for bounding box queries



1 TB dataset, precision 1, took average of 10 randomly located queries

Demo

- **Runs on IOSTACK testbed**
  - 3 Spark and 3 Object Storage nodes
- **Demo dataset**
  - 1 million meters
  - Report every 15 mins
  - 1 day's worth of data
  - 11 GB (csv)
- **Grid Partitioner Config**
  - Cells have precision 1
    - ~10 km^2
  - 100 partitions

Get my neighbours' average usage

```
SELECT AVG(usage)
FROM (
  SELECT vid as meter_id,
                (MAX(index)-MIN(index)) as usage
  FROM dataset
  WHERE (lat BETWEEN 43.300 AND 44.100)
  AND (lng BETWEEN 6.800 AND 7.600)
  GROUP BY vid
)
```

Conclusions: Don’t let your vacation turn into a relocation

- Running SQL queries directly on Big Datasets in Object Storage is viable using metadata
- A small change to Spark enables dramatic performance improvements
- We focused here on geospatial data and the GridPocket use case, although it also applies to other data types and use cases
- IBM is investing in Spark SQL as a backbone for SQL processing services in the cloud

Thanks!

Contact:
paula@il.ibm.com
https://www.linkedin.com/in/paulatashma/

guyger@il.ibm.com
https://www.linkedin.com/in/guy-gerson-82619164/

Backup

GridPocket Use Case and Dataset

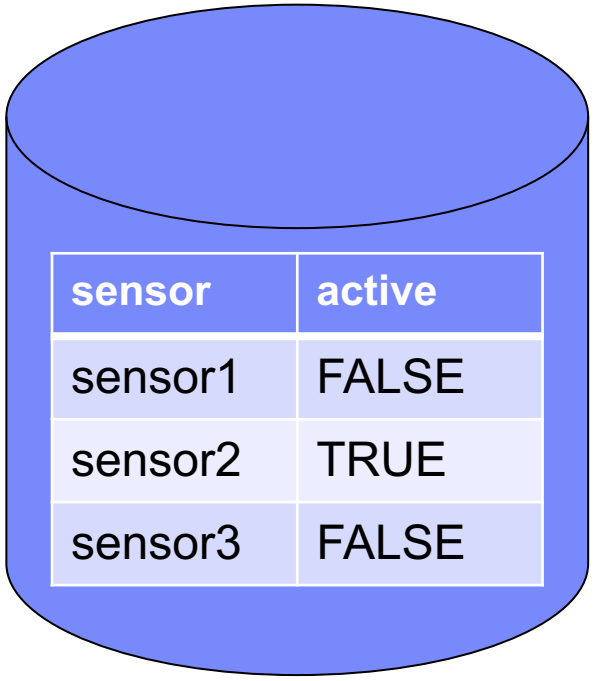
- NoSQL dataset – one large denormalized table
  - **date, index, sumHC, sumHP, type, vid, size, temp, city, region, lat, lng**
  - Index = meter reading
  - sumHC = total of energy consumed since midnight in off-hours
  - sumHP = total of energy consumed since midnight in rush-hours
  - Type = elec/gas/water
  - Vid = meter id
  - Size = apartment size in square meters

Example Filter Scenario

- **Want to analyze data from active sensors only**
- **External DB contains sensor activity info**

Data Layout

Archives/dt=01-01-2014
Archives/dt=01-01-2014/sensor1.json (500MB)
Archives/dt=01-01-2014/sensor2.json (500MB)
Archives/dt=01-01-2014/sensor3.json (500MB)
Archives/dt=02-01-2014
Archives/dt=02-01-2014/sensor1.json (500MB)
Archives/dt=02-01-2014/sensor2.json (500MB)
Archives/dt=02-01-2014/sensor3.json (500MB)
more...



sensor	active
sensor1	FALSE
sensor2	TRUE
sensor3	FALSE

Example Filter

```
class LiveSensorFilter extends ExecutionFileFilter {

 //get a list of live sensors from an external Service
 val activeSensors = SensorService.getLiveSensors

 //returns true if object represents a live sensor
 @Override
 def isRequired(dataFilters:Seq[org.apache.spark.sql.sources.Filter],
                fileStatus: FileStatus) : Boolean = {
   activeSensors.contains(Utils.getSensorIdFromPath(fileStatus))
 }
```

Turn filter on:

```
sqlContext.setConf("spark.sql.execution.fileFilter", “LiveSensorFilter ”)
```