INFORMATIK
BERICHTE

380 – 01/2020

Parallel Trajectory Management in SECONDO

Fabio Valdés, Thomas Behr, Ralf Hartmut Güting

FernUniversität in Hagen

Fakultät für Mathematik und Informatik
D-58084 Hagen

Parallel Trajectory Management in Secondo

Fabio Valdés, Thomas Behr, and Ralf Hartmut Güting

Database Systems for New Applications

Faculty of Mathematics and Computer Science

Fernuniversität Hagen, Germany

January 3, 2020

Abstract

In this document, we present a use case for handling a real-world spatio-temporal dataset on a cluster with the DBMS Secondo. For a set of taxi trajectories recorded in Rome, Italy, we address the raw data import with some data cleaning, distribution on several workers, indexing with different structures, querying the indexed data, and inserting updates.

Contents

1 Introduction

This document provides a stepwise manual for importing, partitioning, indexing, querying, and updating a very large set of raw trajectory data on several machines. Throughout the article, a real-world dataset is applied for demonstration purposes. All listed queries are executed in the freely available database system SECONDO [3, 5, 1].

The dataset considered in the following comprises the trajectories of 318 taxis recorded in the city of Rome during one month [2]. The corresponding file contains almost 22 million timestamped positions together with the taxi identifier, ordered by time, and has a size of 1.5 GBytes. Moreover, we will enlarge the dataset by a factor 10 corresponding to about 220 million time-stamped positions.

For background information on the techniques used in this paper, see the User Manual of SECONDO [6] and the Tutorial on Distributed Query Processing [4].

2 Preparation

To be able to demonstrate the initial loading of data and the setup of the distributed database as well as later updates (when new position data become available), we split the data set into a large part loaded initially and the rest used for an update. Since lines of the file are ordered by time, a simple file split operation suffices to divide data into temporally disjoint ranges.

We assume that the data file `taxi_february.txt` is available in a directory named `cabsrome` right below SECONDO's `bin` directory. We first determine the number of lines in this file, e.g. by a command `wc -l < taxi_february.txt`. There are 21817851 lines.

The Linux command

```
split -l 21000000 -d taxi_february.txt taxidata
```

creates two files `taxidata00` and `taxidata01` with 21000000 and 817851 lines, respectively.

3 Environment

First, the master and worker environment has to be set up. This procedure is detailed in [4], Section 3. We use two different clusters.

The first small cluster has 12 workers besides the master. It uses two desktop computers with the following properties and configuration:

- Computer 1
  - 6 cores, 16 GB main memory, 4 disks, AMD Phenom 1055t six-core processor, running Ubuntu 18.04
  - master using 1 core, one disk, 4 GB memory
  - 4 workers each using one core, 2 GB memory, two workers sharing one disk
- Computer 2
  - 8 cores, 32 GB main memory, 4 disks, AMD Fx-8320 eight-core processor, running Ubuntu 18.04
  - 8 workers each using one core, 3.6 GB memory, two workers sharing one disk

Examples in the text refer to this cluster.

The second, a bit larger, cluster provides 40 workers. It employs five server computers. Each of them has the following configuration:

- 8 cores, 32 GB main memory, 4 disks, Intel Xeon CPU E5-2630, running Ubuntu 18.04
- 8 workers each using one core, 3.6 GB memory, two workers sharing one disk
- master on one of them, using all memory.

Together with the SECONDO commands and queries shown below we indicate running times. The purpose is to give a rough idea of the effort required for this query. Running times are given as pairs, for example

```
9.3 seconds, 1.27 seconds
```

The two entries refer to the small and larger cluster, respectively. Comparing them allows one to get an impression of scalability. If only one time is shown, it refers to the small cluster; the query was not run on the larger cluster.

4 Data Import and Partitioning

After the cluster is prepared, we can begin to import the raw data. The objective is to obtain two different partitionings, one by object identifier and another by the spatio-temporal dimensions, in order to support efficient querying.

The distribution by identifier is rather trivial. Unfortunately, the taxi dataset is not suitable for a standard spatial partitioning, since the trips are rather long compared to the diameter of the whole data space (which roughly equals the city of Rome). For example, a trip may start at one of the airports and end in the city center, hence many trips would be copied to several workers.

Therefore, we apply a small trick and reproduce the data space ten times in a 5×2 raster (other sizes are also possible), in order to support a better spatial partitioning and to also enlarge the dataset.

The resulting 3-dimensional space (2d + time) is then partitioned into $4 \times 2 \times 8$ grid cells. Trajectories are mapped to the cells they intersect. Cells in turn are mapped to the fields of a distributed array.

4.1 Creating a Database

In the first step we create and open a database and restore the `Workers` relation. The number of slots for distributed arrays is set. We also define a port number required by some operators for data exchange which must be reserved for this application.

```
create database taxis;
open database taxis;

restore Workers from Workers;
let NSlots = 40;
let myPort = 1238;
```

We also need to start the monitors controlling the workers:

```
remoteMonitors Cluster40 start
```

These steps assume that a `Workers` relation is available in nested list format in the `secondo/bin` directory as well as a file `Cluster40` specifying the available monitors (see [4]).

4.2 Data Import

The raw data are available as a CSV file `taxidata00` with 21 million lines of the following form (here the first line):

```
156;2014-02-01 00:00:00.739166+01;POINT(41.8836718276551 12.4877775603346)
```

The three attributes are an object identifier, a time stamp, and geographic coordinates, separated by a semicolon. The latter two need to be transformed into a suitable form for SECONDO. The lines are ordered by time stamp.

There are two possible strategies for importing the data into the distributed database:

- Reading the file sequentially on the master and then distribute tuples to workers, or
- Splitting the file into parts, distributing file parts to workers and then reading file parts in parallel by the workers.

The first strategy is more straightforward whereas the second is more scalable. We discuss the two techniques in turn.

4.2.1 Sequential Import and Distribution from the Master

```
let RawDataById =
  [const rel(tuple([Id: int, DateTime: string, Point: string])) value ()]
  csvimport['cabsrome/taxidata00', 0, "", ";", FALSE, FALSE]
  projectextend[Id
  ; Inst: str2instant(substr(.DateTime, 1, 10) + "-" + substr(.DateTime, 12, 23)),
    Pos: makepoint(
      str2real(
        substr(.Point, findFirst(.Point, " ") + 1, findFirst(.Point, ")") - 1)),
      str2real(
        substr(.Point, findFirst(.Point, "(") + 1,  findFirst(.Point, " ") - 1))
  )]
  ddistribute4["RawDataById", hashvalue(.Id, 999997) , 40, Workers]
```

8:49 min, 5:25 min

The operator **csvimport** reads lines from the CSV file, interpreting them according to the schema given by the empty relation argument in front of the operator. Output of the first part of the query

```
[const rel(tuple([Id: int, DateTime: string, Point: string])) value ()]
csvimport['cabsrome/taxi_february.txt', 0, "", ";", FALSE, FALSE]
```

are tuples of the form:

```
      Id : 156
DateTime : 2014-02-01 00:00:00.739166+01
   Point : POINT(41.8836718276551 12.4877775603346)
```

In the next step, the **projectextend** operator keeps the `Id` attribute and computes new attributes `Inst` of type *instant* and `Pos` of type *point*, applying string operations. The resulting tuples have the form:

```
  Id : 156
Inst : 2014-02-01-00:00:00.739
 Pos : point: (12.4877775603346, 41.8836718276551)
```

Finally, the **ddistribute4** operator distributes tuples to 40 slots of a distributed array partitioning by a hash function on the `Id` attribute. Result of the query is a distributed array `RawDataById`.

4.2.2 Parallel Import

Distributing the 21 million lines over 40 slots of a distributed array results in 525000 lines per slot. We can apply a Linux `split` command to divide the large file into 40 parts, each containing 525000 lines:

```
split -l 525000 --numeric-suffixes=10 taxidata00 taxipart
```

This creates 40 files named `taxipart10`, ..., `taxipart49`. We copy these files to the same directory `$HOME/secondo/bin/cabsrome` on all computers running workers, using commands like the following:

```
scp taxipart* ralf@ralf2:/home/ralf/secondo/bin/cabsrome
```

We create a distributed integer array, containing the numbers 0, ..., 39 in its slots. This allows one to control the further processing of 40 slots.

```
let Control40 = createintdarray("Control40", Workers, 40)
```

These three preparatory steps have taken only $12.6 + 14.9 + 2.25 = 29.75$ seconds. The following command imports the data in parallel:

```
let RawDataById2 = Control40
  dmap["",
    [const rel(tuple([Id: int, DateTime: string, Point: string])) value ()]
    csvimport['cabsrome/taxipart' + num2string(. + 10), 0, "", ";", FALSE, FALSE]
    projectextend[Id
    ; Inst: str2instant(substr(.DateTime, 1, 10) + "-" + substr(.DateTime, 12, 23)),
      Pos: makepoint(
        str2real(
          substr(.Point, findFirst(.Point, " ") + 1, findFirst(.Point, ")") - 1)),
        str2real(
          substr(.Point, findFirst(.Point, "(") + 1,  findFirst(.Point, " ") - 1))
    )]
    consume
  ]
  partition["", hashvalue(.Id, 999997), 0]
  collect2["RawDataById2", myPort]
```

3:15 min

In the end, tuples are repartitioned by `Id` with the **partition** and **collect2** operators. Result is a distributed array `RawDataById2`.

4.3 Creating Moving Objects

From the distributed raw data we now create trips associated with object identifiers. Note that for a given object identifier all data points are present in the same slot due to the partitioning by identifiers.

```
let CabsTrips = RawDataById
  dmap["CabsTrips",
    . feed sortby[Id, Inst]
    groupby[Id; TotalTrip: group feed
      approximate[Inst, Pos, TRUE, create_duration(0, 60000)]
      removeNoise[35.0, 2000.0, create_geoid("WGS1984")] ]
    projectextendstream[Id; Trip: .TotalTrip
      sim_trips[create_duration(0, 180000), 100.0, create_geoid("WGS1984")] ]
    filter[no_components(.Trip) > 1]
    consume
  ]
```

1:48 min, 14.6 seconds

In the invocation of the **approximate** operator, which combines timestamps and positions into complete trajectories, we assume that a continuous movement ends if two consecutive data points for the same taxi identifier have a temporal distance of more than one minute (note that the command `create_duration(0, 60000)` creates an object of the type *duration* having a length of 60,000 milliseconds, i.e., one minute).

To the trajectories (moving points) built by the **approximate** operator, a further operator **removeNoise** is applied. GPS data sometimes contain erroneous positions that can to some extent be recognized by lying far away from the other observation points within a temporal sequence of observations. This results in the creation of two adjacent movement units that have a large speed and distance traversed. The **removeNoise** operator discovers such pairs of subsequent units with speed and/or distance higher than a specified threshold and replaces them by a single unit (so that the erroneous point is ignored). For the current dataset we assume that taxis do not move at a speed higher than 120 km/h or traverse between two observations a distance higher than 2000 meters. The latter distance can be traversed within one minute by a car moving at 120 km/h; this is consistent with the parameter of approximate which separates trips if there is a gap of more than one minute between observations. The two parameters of **removeNoise** are the speed in meters per second (120 km/h $\approx$ 35 m/s) and the distance in meters if a geoid is given as a third parameter.

The **sim_trips** operator splits trajectories into shorter trips whenever there are definition gaps, i.e., adjacent positions not connected by movement. It also splits them at stationary intervals with a duration of at least three minutes, where an interval is viewed as stationary if its speed is below 100 meters per hour. Finally, only trips having at least two units are kept.

4.4 Looking at the Dataset

We can look at the result and check for plausibility:

```
query CabsTrips dmap["", . count] getValue
```

Here the **getValue** operator copies a distributed array into a local array on the master. The result looks like this:

```
*************** BEGIN ARRAY ***************
--------------- Field No: 0 ---------------
9066
--------------- Field No: 1 ---------------
8957
--------------- Field No: 2 ---------------
9970
--------------- Field No: 3 ---------------
8400


...

--------------- Field No: 39 ---------------
8500
***************  END ARRAY  ***************
```

So there are about 9000 trips per slot, the total is given by

```
query CabsTrips dmap["", . count] getValue tie[. + ..]
```

and is 337182. The **tie** operator takes a binary parameter function to combine any two adjacent fields of an array, here addition, and so computes an aggregate over an array. The query could also be written as

```
query CabsTrips dmap["", . count] getValue tie[fun(x:int, y:int) x + y]
```

The average number of units (observations) per trip is

```
query CabsTrips dmap["", . feed projectextend[; N: no_components(.Trip)] sum[N]]
  dsummarize transformstream sum[Elem]
  / 337182
```

which is about 40.

Here the **dsummarize** operator is applied to the distributed integer array resulting from the **dmap** operation and feeds these integers into a stream of values on the master. This is further transformed into a stream of tuples (with attribute `Elem`) to which the **sum** aggregation is applied.

We determine for each trip its traversed distance in meters and total time in minutes:

```
let CabsTripsDistDur = CabsTrips
  dmap["CabsTripsDistDur", . feed
    projectextend[
    ; Distance: val(final(distancetraversed(.Trip, create_geoid("WGS1984")))),
      Duration: get_duration(deftime(.Trip)) / [const duration value (0 60000)]]
  ]
```

11.65 sec

Note that the division of durations returns an integer result (like integer division), hence a result of 0 indicates a duration of less than one minute.

Let us see from each slot a few example tuples:

```
query CabsTripsDistDur dmap["", . feed head[3] consume] getValue
```

The result is:

```
*************** BEGIN ARRAY ***************
--------------- Field No: 0 ---------------

Distance : 17.1818541061
Duration : 0

Distance : 262.1977324963
Duration : 0

Distance : 3866.498835861
Duration : 9

--------------- Field No: 1 ---------------

Distance : 3877.4149047889
Duration : 15

Distance : 9662.5241977585
Duration : 41

Distance : 7231.7033269336
Duration : 10

--------------- Field No: 2 ---------------

Distance : 5565.8362844199
Duration : 20

...
```

The minimal, maximal, and average distance traversed and duration of trips, respectively, can be computed as

```
query CabsTripsDistDur dmap["", . feed
    groupby[
    ; MinDist: group feed min[Distance],
      MaxDist: group feed max[Distance],
      AvgDist: group feed avg[Distance],
      MinDur: group feed min[Duration],
      MaxDur: group feed max[Duration],
      AvgDur: group feed avg[Duration]]
  ]
  getValue
```

with result

```
*************** BEGIN ARRAY ***************
--------------- Field No: 0 ---------------

MinDist : 4.08e-07
MaxDist : 31500.5420808775
AvgDist : 1889.0556102962
 MinDur : 0
 MaxDur : 183
 AvgDur : 7.0586807854

--------------- Field No: 1 ---------------

MinDist : 2.909e-07
MaxDist : 49709.1580841572
AvgDist : 1702.7162155473
 MinDur : 0
 MaxDur : 209
 AvgDur : 6.7019091214

...
```

Finally, let us see the spatial bounding box of trips for each slot at the graphical user interface.

```
query CabsTrips
  dmap["", . feed projectextend[; Box: bbox2d(.Trip)] transformstream
    collect_box[TRUE]]
  getValue
```

The result is shown in Figure 1.

4.5 Multiplying the Dataset

In this step we enlarge the dataset by creating a number of copies shifted spatially and with numerically shifted identifiers. To prepare this, we first need to know the maximal identifier and the width and height of the spatial bounding box of all points.

The maximal identifier can be determined by the following query:

```
query CabsTrips dmap["", . feed max[Id]] dsummarize transformstream max[Elem]
```

2.7 sec

The maximal identifier is 374.

The spatial bounding box of all trips (GPS positions) and its width and height can be determined as follows:

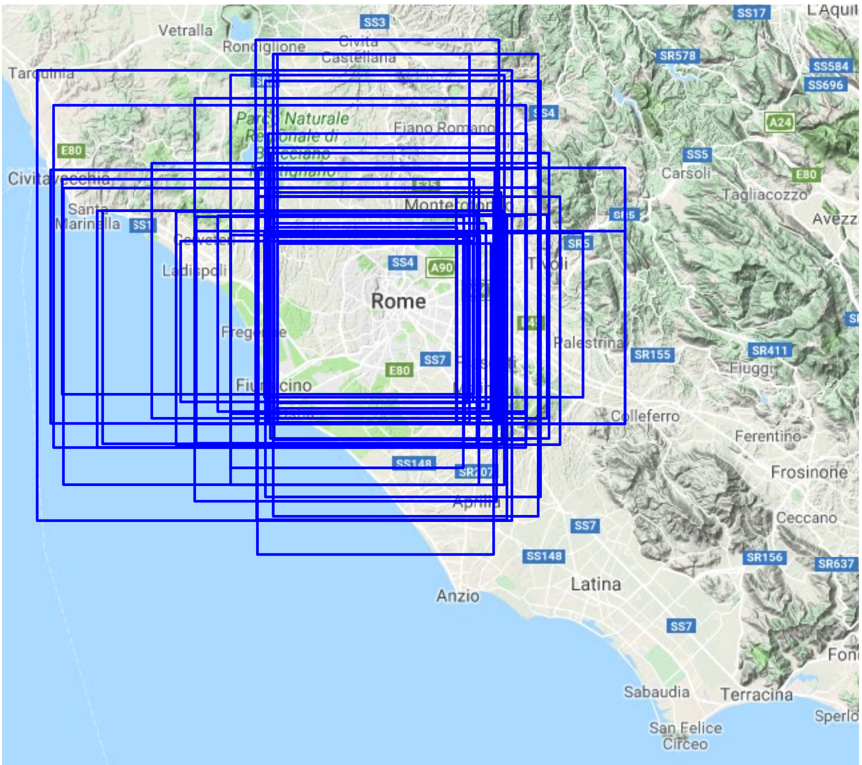

Figure 1: Bounding Boxes of Trips for Each Slot of the Distributed Array

```
let bboxR = CabsTrips
  dmap["", . feed projectextend[; Box: bbox(.Trip)] transformstream
    collect_box[TRUE]]
  dsummarize collect_box[TRUE]

let deltax = maxD(bboxR, 1) - minD(bboxR, 1);
let deltay = maxD(bboxR, 2) - minD(bboxR, 2)
```

The `deltax` and `deltay` values will be needed by the workers; so we make these values available to them.

```
query share("deltax", TRUE, Workers);
query share("deltay", TRUE, Workers);
```

We now have a choice of how many copies of the dataset we wish to create. In this example, we create 10 copies arranged in a 5×2 grid. That is, the area of the original bounding box is repeated four times in x- and once in y-direction. For the identifiers, we reserve disjoint ranges of 1000 values for the 10 copies. The resulting enlarged dataset is distributed again by `Id`.

```
let CabsId = CabsTrips
  dmap["", . feed
    loopsel[fun(t: TUPLE)
      intstream(0, 4) namedtransformstream[Nx]
      intstream(0, 1) namedtransformstream[Ny]
      product
      projectextend[
      ; Id: attr(t, Id) + ((5 * .Ny) + .Nx) * 1000,
        Trip: attr(t, Trip)
          translate[[const duration value (0 0)], .Nx * deltax, .Ny * deltay]]
    ]
  ]
```

```
    partition["", hashvalue(.Id, 999997), 0]
    collect2["", myPort]
    dmap["CabsId", . feed consume]
```

11:27 min, 2:17 min

The **loopsel** operator evaluates for each tuple in its argument stream a parameter function which returns a stream of tuples; it returns the concatenation of all these streams. Here for each argument tuple we create the 10 copies with shifted `Id` and `Trip` attributes. After repartitioning, the final application of the **dmap** operator serves to transform the distributed file array returned by **collect2** into a distributed array of relations. We need a *darray* to be able to build indexes over its relations.

Again we check the result for plausibility.

```
query CabsId dmap["", . count] getValue tie[. + ..]
```

The result is 3371820.

We compare the numbers of units in the original and the multiplied data set:

```
query CabsTrips dmap["", . feed projectextend[; N: no_components(.Trip)] sum[N]]
  dsummarize transformstream sum[Elem]

query CabsId
  dmap["", . feed projectextend[; N: no_components(.Trip)] sum[N]]
  dsummarize transformstream sum[Elem]
```

The results are 13460706 and 134607060, respectively.

The maximal identifier is

```
query CabsId dmap["", . feed max[Id]] dsummarize transformstream max[Elem]
```

returning 9374 as it should be. Finally, we look at the bounding boxes of slots

```
query CabsId
  dmap["", . feed projectextend[; Box: bbox2d(.Trip)] transformstream
    collect_box[TRUE]]
  dsummarize consume
```

and obtain the red boxes in Figure 2 as compared to the blue boxes of the original dataset.

4.6 Partitioning

Partitioning determines which objects are co-located within the same slot and hence stored on the same worker. This is relevant for keeping operations restricted to one or a few workers, rather than all, and especially to support joins over partitioning attributes.

We propose to maintain two copies of the dataset, one partitioned by object identifier, the other spatio-temporally.

4.6.1 Partitioning by Object Identifier

The distributed array `CabsId` is already partitioned by object identifier as this was the last step in the query creating it. So all trips of a given object are co-located.

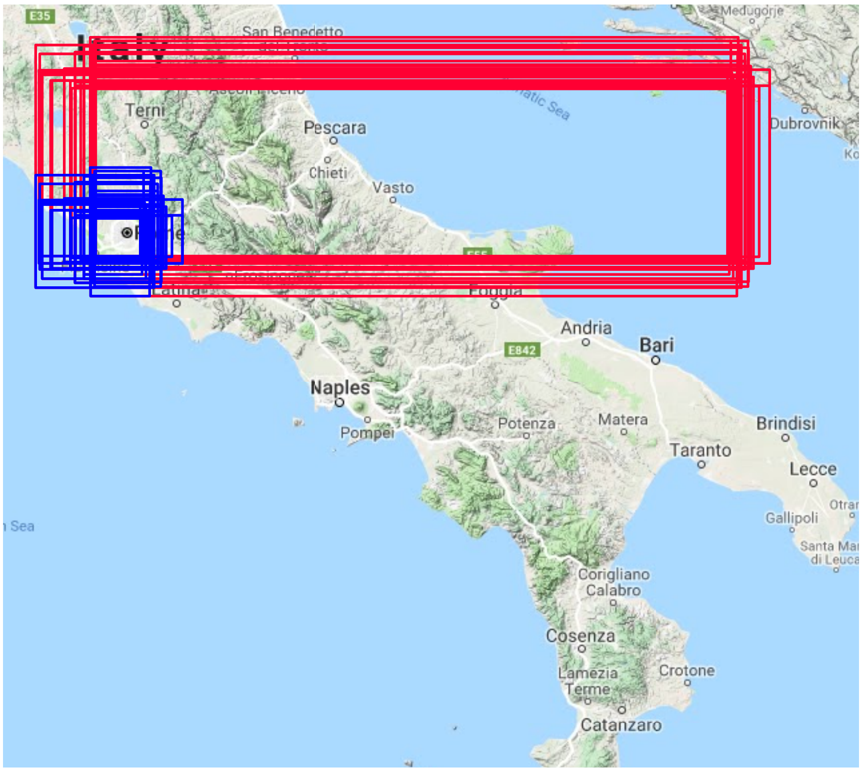

Figure 2: Bounding Boxes of Trips for Each Slot of the Original (blue) and the Multiplied Dataset (red)

4.6.2 Spatio-temporal Partitioning

To design the spatio-temporal partitioning, we need the 3d bounding box of the data, that is, x- and y-coordinates and the temporal extent. It is given by the query

```
let bboxR10 = CabsId
  dmap["", . feed projectextend[; Box: bbox(.Trip)] transformstream
    collect_box[TRUE]]
  dsummarize collect_box[TRUE]
```

6.65 seconds, 1.6 seconds

Note that here we do not apply the **bbox2d**, but the **bbox** operator which constructs a 3d bounding box. The resulting rectangle has the value

```
rect: ( (11.753154551,41.513268021,5143.0000086) -
  (17.804743105,43.090930714,5171.6134439) )
```

represented in the form $((x_{bottom}, y_{bottom}, t_{bottom}) - (x_{top}, y_{top}, t_{top}))$. Instants are represented by real numbers by counting the number of days (and fractions thereof) since the NULL date 2000-01-03. We can convert a *real* into an *instant* by the operator **create_instant**, hence by the queries

```
query create_instant(5143.0000086);
query create_instant(5171.6134439)
```

we find that the temporal range of the dataset loaded is:

```
2014-02-01-00:00:00.743
2014-03-01-14:43:21.553
```

We enlarge the data space a bit introducing an x-range 11.0 - 19.0, a y-range 41.0 - 44.0 and a t-range 5140.0 - 5180.0. Within this space we decide, somewhat arbitrarily, to create a grid of $4 \times 2 \times 8$ cells. The grid data structure is created as follows:

```
let grid = createCellGrid3D(11.0, 41.0, 5140.0, 2.0, 1.5, 5.0, 4, 2)
```

The first three parameters define the origin of the data space in x, y, t, the next three the respective cell widths. The last two parameters define the numbers of cells in x- and y-direction. The grid is unlimited in temporal direction. The purpose of the grid is to assign a continuous range of cell numbers starting from 1 that can be used for partitioning. Cells are numbered first in increasing x-, then in y- and last in t-direction.

The grid needs to be made available to the workers:

```
query share("grid", TRUE, Workers)
```

The spatio-temporally partitioned copy of the data set is built as follows:

```
let CabsST = CabsId
  partitionF["", . feed
    extendstream[Cell: cellnumber(bbox(.Trip), grid)]
    extend[Box: bbox(.Trip)],
    ..Cell, NSlots]
  collect2["", myPort]
  dmap["CabsST", . feed consume]
```

20:27 min, 3:37 min

Here the **partitionF** operator allows one to first apply a function to the relation in each slot, returning a (modified) tuple stream, and then to partition the resulting stream of tuples by one of its attributes.

Within the function, the bounding box of each trip is compared to the grid structure by the **cellnumber** operator; it returns a stream of integer values, the cell numbers of cells intersected by the bounding box. The **extendstream** operator creates one copy of the input tuple for each returned value, the latter is kept in the new attribute `Cell`.

The following application of the **extend** operator adds to each tuple an `Original` attribute. It is `true` if the cell `Cell` value is the first emitted by the **cellnumber** operator. So this tuple is considered the original, any tuples with following cell numbers as copies. This can be used to efficiently eliminate duplicates from query results.

The last parameter of the **partitionF** operator gives the number of slots of the partitioning to be created. Here the original data should fill slots up to 64; we leave space for future tuples by creating 120 slots. [1] Slot numbers are generally mapped round-robin to workers. Therefore full and initially empty slots should be distributed evenly among workers.

Finally, the distributed file array is transformed into a distributed array to have relations in each slot that can be indexed.

Again, we examine the result a bit:

```
query CabsST dmap["", . count] getValue
```

The result looks like this:

```
--------------- Field No: 0 ---------------
0
--------------- Field No: 1 ---------------
7948
--------------- Field No: 2 ---------------
15783
```

[1] If cell numbers larger than 120 occur in the future, they will be taken modulo 120 and so be mapped to the first slots again.

```
--------------- Field No: 3 ---------------
8198
--------------- Field No: 4 ---------------
7947

...
--------------- Field No: 55 ---------------
22857
--------------- Field No: 56 ---------------
22162
--------------- Field No: 57 ---------------
0
--------------- Field No: 58 ---------------
0
--------------- Field No: 59 ---------------
0

...
--------------- Field No: 119 ---------------
0
***************  END ARRAY  ***************
```

Hence the last filled cell is numbered 56. The total number of trips is given by

```
query CabsST dmap["", . count] getValue tie[. + ..]
```

returning 3380464. Note that for `CabsId` this number was 3371820. The difference of 8644 is due to the fact that in `CabsST` some trips intersecting cell boundaries are duplicated. But these are relatively few. If we restrict to originals, as in

```
query CabsST dmap["", . feed filter[.Original] count] getValue tie[. + ..]
```

we get indeed the result 3371820.

The maximal number of trips in a cell is

```
query CabsST dmap["", . count] dsummarize transformstream max[Elem]
```

returning 204663.

We can visualize the distribution of trips over cells (Figure 3) in the Javagui by the following query:

```
query CabsST dmap["", . count] dsummarize transformstream addcounter[N, 0]
  extend[Bar: rectangle2(.N * 10000, (.N * 10000) + 8000, 0, .Elem)] consume
```

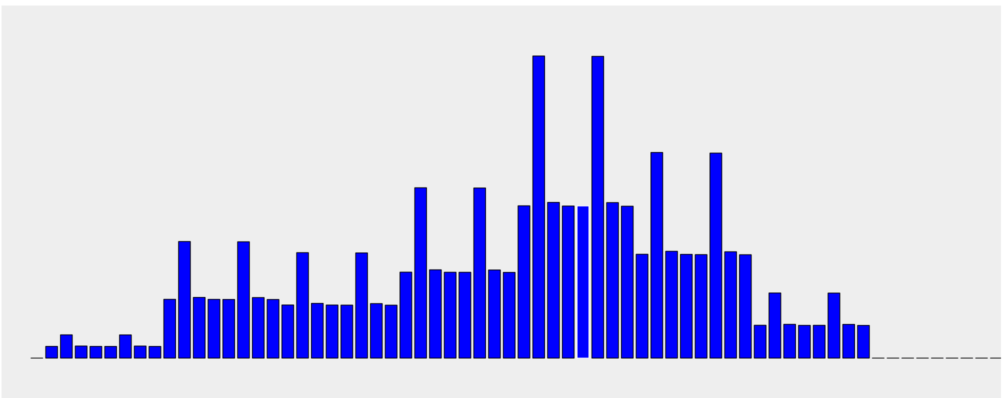

Figure 3: Distribution of Trips Over Slots for `CabsST` (empty slots to the right omitted)

5 Indexing

We wish to support two types of queries:

1. Find all trips for a given object identifier.
2. Retrieve all trips intersecting a given spatio-temporal range.

To enable efficient execution, we create B-trees over taxi identifiers and R-trees over trip data.

5.1 By Identifier

We create a B-tree for indexing the taxi identifiers over the `CabsId` distributed array.

```
let CabsId_Id_btree = CabsId dmap["CabsId_Id_btree", . createbtree[Id]]
```

11.4 seconds, 1.4 seconds

5.2 By Spatio-temporal Information

The applied index structure is a three-dimensional R-tree.

```
let CabsST_Trip_rtree = CabsST
  dmap["CabsST_Trip_rtree",  . feed addid
    extend[ScaledBox: scalerect(.Box, 1000000.0, 1000000.0, 1000.0)]
    sortby[ScaledBox]
    remove[ScaledBox]
    bulkloadrtree[Box]
  ]
```

20.7 seconds, 4.6 seconds

Here the **bulkloadrtree** operator processes a stream of tuples containing a tuple identifier attribute (allowing for direct access to the tuple in the relation) and a 3d rectangle to be used as index entry. This operator packs arriving entries sequentially into pages; it requires to receive a stream of tuples in z-order. To this end, the arriving stream is sorted by a numerically enlarged box. Enlarging the box is necessary because the sorting by rectangles uses only the integer part of coordinates; for geographic coordinates this is not precise enough. When the scaled box has been used for sorting, it can be disposed again.

This R-tree is built over bounding boxes of entire trips. Such boxes cover relatively large areas. A spatio-temporal range query with a small range may retrieve many trips whose boxes overlap the range. To have a more fine-grained indexing, we can alternatively create an index over units of trips.

```
let CabsST_UTrip_rtree = CabsST
  dmap["CabsST_UTrip_rtree",  . feed addid
    projectextendstream[TID; UTrip: units(.Trip)]
    extend[Box: bbox(.UTrip)]
    extend[ScaledBox: scalerect(.Box, 1000000.0, 1000000.0, 1000.0)]
    sortby[ScaledBox]
    remove[ScaledBox]
    bulkloadrtree[Box]
  ]
```

25:04 min

The **units** operator returns a stream of the units (linear pieces of movement) of a trajectory. The **projectextendstream** operator keeps of each incoming tuple the `TID` attribute and produces as many copies of the tuple as there are units. Hence the resulting stream consists of pairs of a `TID` and a unit. This is fed into the index construction as before.

Building this set of indexes takes much longer because in total about 135 million index entries are created.

6 Querying

Querying by identifier, time interval, or spatial property can now be performed efficiently.

6.1 By Identifier

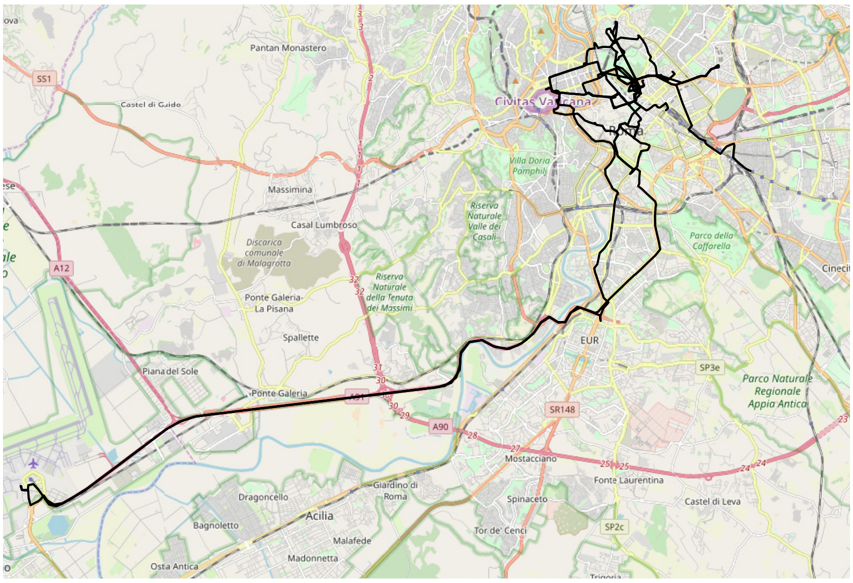

Figure 4: Query result showing all trips corresponding to the identifier 11

The following query finds and outputs all trips corresponding to the identifier 11, as depicted in Figure 4.

```
query CabsId_Id_btree CabsId
  dmap2["", . .. exactmatch[11], myPort]
  dsummarize consume
```

14.3 seconds, 11.5 seconds

6.2 By Spatio-temporal Dimension

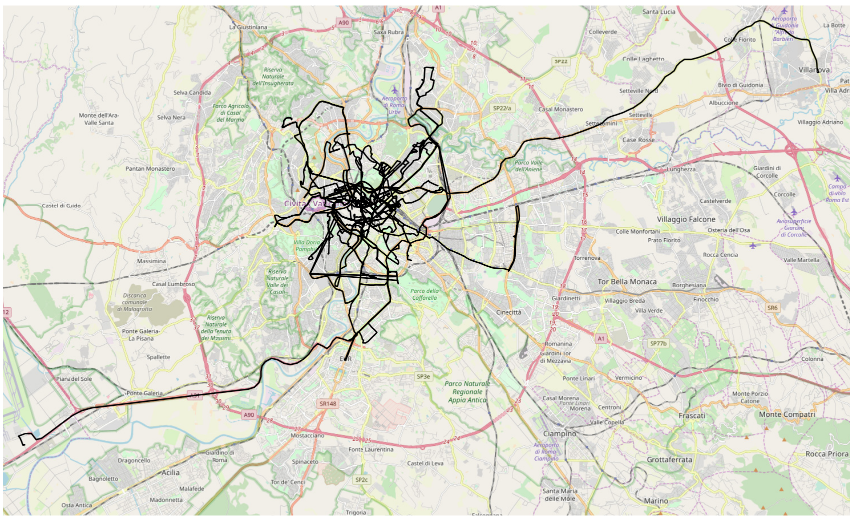

Figure 5: Query result showing all trips corresponding to the query object `qObject`

The following query finds all trips passing the city center of Rome on February 1, 2014, between 7 and 8 pm. It is executed after defining and sharing the query object. The result is shown in Figure 5.

```
let qObject = rectangle3(12.464446, 12.5013, 41.879938, 41.911759,
  instant2real(theInstant(2014, 02, 01, 19)),
  instant2real(theInstant(2014, 02, 01, 20)));

query share("qObject", TRUE, Workers)

query CabsST_Trip_rtree CabsST
  dmap2["", . .. windowintersects[qObject]
    filter[.Trip passes rectproject(qObject, 1, 2) rect2region]
    filter[.Trip present rect2periods(qObject)], myPort]
  dsummarize sort rdup consume
```

22.7 seconds

Here the **windowintersects** operator retrieves from the R-tree `CabsST_Trip_rtree` the trips whose 3d bounding box intersects the query box `qObject`. The following two **filter** steps check that the actual trajectory intersects the query box in space and time.

Note that a query box may overlap several grid cells. The same holds for a data box, i.e., the bounding box of a trip. Therefore query box and data box may meet in different cells of the grid, hence in different slots of the distributed array, and we may have duplicate results.

One way to get rid of them is to discover them on the master. This is done in the query above with the **sort** and **rdup** operators. However, this method may be expensive for large result sets due to the necessary sorting. The **rdup** operator just removes consecutive equal tuples in a tuple stream.

Another method is to decide within each cell or slot whether this result instance is to be reported. The idea is to compute a point, the smallest point within the intersection of query rectangle and data rectangle. This point can lie in only one cell. Only for this cell the result will be reported. This method is implemented in the following variant of the query:

```
query CabsST_Trip_rtree CabsST
  dmap2["", . .. windowintersects[qObject]
    filter[gridintersects(grid, qObject, bbox(.Trip), .Cell)]
    filter[.Trip passes rectproject(qObject, 1, 2) rect2region]
    filter[.Trip present rect2periods(qObject)],
  myPort]
  dsummarize consume
```

9.7 seconds, 9.6 seconds

Here the **gridintersects** operator checks whether the smallest intersection point of the `qObject` and the bounding box of the trip lies in the cell numbered `.Cell` with respect to the argument grid.

The queries shown in this section are a bit slow. This is because within the **dmap2** operation, the respective query is evaluated for each slot, even though for almost all slots the result will be empty. For the query

```
query CabsId_Id_btree CabsId
  dmap2["", . .. exactmatch[11], myPort]
  dsummarize consume
```

it would be sufficient to evaluate the slot to which the Id 11 is mapped. For the spatio-temporal query we could restrict evaluation to the slots whose grid cells intersect the query object. We can easily determine which slots are concerned, by the queries

```
query hashvalue(11, 999997) mod NSlots;
query cellnumber(qObject, grid) use[. mod NSlots] consume
```

Fortunately there exist variants of the **dmap** family of operators called **pdmap** (partial dmap). These operators take an additional argument (written first) which is a stream of slot numbers that need to be evaluated; other slots are ignored. With these operators, the queries can be written as follows:

```
query hashvalue(11, 999997) feed use[. mod NSlots] CabsId_Id_btree CabsId
  pdmap2["", . .. exactmatch[11], myPort]
  dsummarize consume
```

10.1 seconds, 11.0 seconds

```
query cellnumber(qObject, grid) use[. mod NSlots] CabsST_Trip_rtree CabsST
  pdmap2["", . .. windowintersects[qObject]
    filter[gridintersects(grid, qObject, bbox(.Trip), .Cell)]
    filter[.Trip passes rectproject(qObject, 1, 2) rect2region]
    filter[.Trip present rect2periods(qObject)], myPort]
  dsummarize consume
```

7.1 seconds, 8.2 seconds

For these queries, improvements are not large, possibly because the main effort is in copying data.

7 Updates

We assume that after setting up the database, it is occasionally extended by new recently collected position data. This means, the recently collected positions need to be transformed into moving points (data type *mpoint*) and the new data be inserted into distributed relations and indexes. Furthermore, continuous trips crossing the "update boundary", that is, comprising positions before and after an update, should be recovered. That is, the respective existing and the new trip should be merged into one.

We demonstrate this with the left-over positions from Section 2 in file `taxidata01`. They first need to be processed like the initial data set. Here, we choose the method of distributing data from the master.

7.1 Data Import

```
let RawDataByIdUpdate =
  [const rel(tuple([Id: int, DateTime: string, Point: string])) value ()]
  csvimport['cabsrome/taxidata01', 0, "", ";", FALSE, FALSE]
  projectextend[Id
  ; Inst: str2instant(substr(.DateTime, 1, 10) + "-" + substr(.DateTime, 12, 23)),
    Pos: makepoint(
      str2real(
        substr(.Point, findFirst(.Point, " ") + 1, findFirst(.Point, ")") - 1)),
      str2real(
        substr(.Point, findFirst(.Point, "(") + 1,  findFirst(.Point, " ") - 1))
  )]
  ddistribute4["RawDataByIdUpdate", hashvalue(.Id, 999997) , NSlots, Workers]
```

26.9 seconds, 14.2 seconds

7.2 Creating Moving Objects

```
let CabsTripsUpdate = RawDataByIdUpdate
  dmap["CabsTripsUpdate",
    . feed sortby[Id, Inst]
    groupby[Id; TotalTrip: group feed
      approximate[Inst, Pos, TRUE, create_duration(0, 60000)]
      removeNoise[35.0, 2000.0, create_geoid("WGS1984")] ]
    projectextendstream[Id; Trip: .TotalTrip
      sim_trips[create_duration(0, 180000), 100.0, create_geoid("WGS1984")] ]
    filter[no_components(.Trip) > 1]
    consume
  ]
```

10.4 seconds, 2.6 seconds

7.3 Multiplying the Dataset

In the initial setup of the database we multiplied the dataset creating copies laid out in a 5×2 grid. We follow the same logic for the updates.

```
let CabsIdUpdate = CabsTripsUpdate
  dmap["", . feed
    loopsel[fun(t: TUPLE)
      intstream(0, 4) namedtransformstream[Nx]
      intstream(0, 1) namedtransformstream[Ny]
      product
      projectextend[
      ; Id: attr(t, Id) + ((5 * .Ny) + .Nx) * 1000,
        Trip: attr(t, Trip)
          translate[[const duration value (0 0)], .Nx * deltax, .Ny * deltay]]
    ]
  ]
  partition["", hashvalue(.Id, 999997), 0]
  collect2["", myPort]
  dmap["CabsIdUpdate", . feed consume]
```

1:12 min, 18.4 seconds

7.4 Updating `CabsId`

We now need to update the two distributed versions of the dataset, `CabsId` and `CabsST`, partitioned by Id and spatio-temporally, including their indexes.

The update relation `CabsIdUpdate` has been distributed in the same way as `CabsId`; hence its tuples can now be inserted locally into `CabsId`.

```
query CabsIdUpdate CabsId CabsId_Id_btree
  dmap3["", $1 feed $2 insert $3 insertbtree[Id] count, myPort] getValue
```

20.85 seconds, 4.4 seconds

Here for each slot, the stream of tuples obtained from `CabsIdUpdate` is inserted into the relation in `CabsId`. The **insert** operator outputs the same stream extended by tuple identifiers which are used for creating index entries by the **insertbtree** operator. This operator also passes the stream of tuples to its successor operator (possibly to be used for further index updates). Here it is the **count** operator which finally collects the stream of tuples.

We now need to merge trips crossing the update boundary. To be consistent with the creation of trajectory data described in Section 4.3, the following rules should be observed:

- Two positions with a time difference of less than 60 seconds are connected by linear movement by the **approximate** operator. Hence two trips with the same object identifier should be concatenated if the time difference between the final point of the first trip and the initial point of the second trip is less than 60 seconds. In this case, there should be a unit connecting the final with the initial point.
- A trajectory (*mpoint*) is split by the **sim_trips** operator at definition gaps. A definition gap occurs, if the time difference between the final and the initial point is longer than 60 seconds. In this case the two trips do not need to be merged.
- The **sim_trips** operator also splits trajectories at stationary intervals of at least three minutes duration. However, if we connect the two trips according to the first rule, a unit between the final and the initial point is created of a duration of less than 60 seconds. Hence this behaviour of the **sim_trips** operator is not relevant here.

In summary, we just need to connect trips if final and initial points are less than 60 seconds apart.

Matching arbitrary pairs of trips on equal identifiers and this condition would be expensive. Fortunately we can restrict to trips whose final or initial position is close to the update boundary. We first compute this instant, calling it `UpdateStart`. The value is contained in the first tuple of the positions read from `taxidata01`.

```
let UpdateStart =
  [const rel(tuple([Id: int, DateTime: string, Point: string])) value ()]
  csvimport['cabsrome/taxidata01', 0, "", ";", FALSE, FALSE]
  projectextend[Id
  ; Inst: str2instant(substr(.DateTime, 1, 10) + "-" + substr(.DateTime, 12, 23)),
    Pos: makepoint(
      str2real(
        substr(.Point, findFirst(.Point, " ") + 1, findFirst(.Point, ")") - 1)),
      str2real(
        substr(.Point, findFirst(.Point, "(") + 1,  findFirst(.Point, " ") - 1))
  )] extract[Inst]

query share("UpdateStart", TRUE, Workers)
```

The following query computes the pairs of trips to be connected.

```
let UpdatePairsId = CabsId CabsId
  dmap2["UpdatePairsId", . feed addid
    filter[UpdateStart > inst(final(.Trip))]
    filter[(UpdateStart - inst(final(.Trip)))
      < create_duration(0, 60000)] c1
    .. feed addid
      filter[UpdateStart < inst(initial(.Trip))]
      filter[(inst(initial(.Trip)) - UpdateStart)
      < create_duration(0, 60000)] c2
    itHashJoin[Id_c1, Id_c2]
    filter[
      (inst(initial(.Trip_c2)) - inst(final(.Trip_c1)))
        < create_duration(0, 60000)] consume,
  myPort]
```

1:17 min, 5.5 seconds

We insert concatenated trips into the set `CabsId`, also updating the index `CabsId_Id_btree`. Concatenated trips are formed by streaming the units of the first trip, then a single unit providing the connection, and then the units of the second trip. The stream of units is collected into an *mpoint* value by the **makemvalue** operator.

```
query UpdatePairsId CabsId CabsId_Id_btree
  dmap3["", . feed
    projectextend[
    ; Id: .Id_c1,
      Trip:
        units(.Trip_c1)
        the_unit(val(final(.Trip_c1)), val(initial(.Trip_c2)),
          inst(final(.Trip_c1)), inst(initial(.Trip_c2)),
          TRUE, FALSE) feed concat
        units(.Trip_c2) concat transformstream
        makemvalue[Elem]
    ]
    $2 insert
    $3 insertbtree[Id]
    count,
  myPort] getValue
```

6.0 seconds, 2.76 seconds
We then delete the trips contained in the set `UpdatePairsId` from `CabsId` and `CabsId_Id_btree`.

```
query UpdatePairsId CabsId CabsId_Id_btree
  dmap3["",
    . feed projectextend[; TID: .TID_c1]
    . feed projectextend[; TID: .TID_c2]
    concat
    $2 deletebyid4[TID]
    $3 deletebtree[Id]
    count,
  myPort] getValue
```

6.5 seconds, 3.5 seconds

7.5 Updating `CabsST`

We apply the same update strategy to the set `CabsST`. First, the new trips need to be distributed spatio-temporally.

```
let CabsSTUpdate = CabsIdUpdate
  partitionF["", . feed
    extendstream[Cell: cellnumber(bbox(.Trip), grid)]
    extend[Box: bbox(.Trip)],
    ..Cell, NSlots]
  collect2["", myPort]
  dmap["CabsSTUpdate", . feed consume]
```

155.1 seconds, 17.6 seconds
The new trips are inserted into `CabsST` and `CabsST_Trip_rtree`.

```
query CabsSTUpdate CabsST CabsST_Trip_rtree
  dmap3["", . feed $2 insert $3 insertrtree[Box] count, myPort] getValue
```

1:17 min, 11:25 seconds
Now we find matching pairs of trips within the spatio-temporal partitioning.

```
let UpdatePairsST = CabsST CabsST
  dmap2["UpdatePairsST", . feed addid
    filter[UpdateStart > inst(final(.Trip))]
    filter[(UpdateStart - inst(final(.Trip)))
```

```
      < create_duration(0, 60000)] c1
    .. feed addid
      filter[UpdateStart < inst(initial(.Trip))]
      filter[(inst(initial(.Trip)) - UpdateStart)
      < create_duration(0, 60000)] c2
    itHashJoin[Id_c1, Id_c2]
    filter[.Cell_c1 = .Cell_c2]
    filter[
      (inst(initial(.Trip_c2)) - inst(final(.Trip_c1)))
        < create_duration(0, 60000)] consume,
  myPort]
```

1:45 min, 6.8 seconds
We insert concatenated trips into the set `CabsST`, also updating the index `CabsId_ST_rtree`.

```
query UpdatePairsST CabsST CabsST_Trip_rtree
  dmap3["", . feed
    projectextend[
    ; Id: .Id_c1,
      Trip:
        units(.Trip_c1)
        the_unit(val(final(.Trip_c1)), val(initial(.Trip_c2)),
          inst(final(.Trip_c1)), inst(initial(.Trip_c2)),
          TRUE, FALSE) feed concat
        units(.Trip_c2) concat transformstream
        makemvalue[Elem],
      Cell: .Cell_c1
    ]
    extend[Box: bbox(.Trip)]
    $2 insert
    $3 insertrtree[Box]
    count,
  myPort] getValue
```

5.9 seconds, 2.2 seconds
Finally, we delete the trips contained in `UpdatePairsST` from `CabsST` and `CabsST_Trip_rtree`.

```
query UpdatePairsST CabsST CabsST_Trip_rtree
  dmap3["",
    . feed projectextend[; TID: .TID_c1]
    . feed projectextend[; TID: .TID_c2]
    concat
    $2 deletebyid4[TID]
    $3 deletertree[Box]
    count,
  myPort] getValue
```

6.5 seconds, 2.8 seconds
This concludes processing the updates.

References

[1] SECONDO – an extensible database system. `http://dna.fernuni-hagen.de/Secondo.html/index.html`, 2019.

[2] L. Bracciale, M. Bonola, P. Loreti, G. Bianchi, R. Amici, and A. Rabuffi. Crawdad dataset roma/taxi. `http://crawdad.org/roma/taxi/20140717`, 2014.

[3] V. T. de Almeida, R. H. Güting, and T. Behr. Querying moving objects in SECONDO. In *MDM*, page 47, 2006.

[4] R. H. Güting and T. Behr. Tutorial: Distributed Query Processing in SECONDO. `http://dna.fernuni-hagen.de/Secondo.html/files/Documentation/General/DistributedQueryProcessinginSecondo.pdf`, 2019.

[5] R. H. Güting, T. Behr, and C. Düntgen. SECONDO: A platform for moving objects database research and for publishing and integrating research implementations. *IEEE Data Eng. Bull.*, 33(2):56–63, 2010.

[6] R. H. Güting, F. Valdés, H. H. Hennings, J. K. Nidzwetzki, F. Heinz, and T. Behr. SECONDO User Manual, Version 4.2. `http://dna.fernuni-hagen.de/Secondo.html/files/Documentation/General/SecondoManual.pdf`, 2019.

Verzeichnis der zuletzt erschienenen Informatik-Berichte

[369] Güting, R.H., Valdés, F., Damiani, M.L.:
Symbolic Trajectories, 12/2013

[370] Bortfeldt, A., Hahn, T., Männel, D., Mönch, L.:
Metaheuristics for the Vehicle Routing Problem with Clustered Backhauls and 3D Loading Constraints, 8/2014

[371] Güting, R. H., Nidzwetzki, J. K.:
DISTRIBUTED SECONDO: An extensible highly available and scalable database management system, 5/2016

[372] M. Kulaš
A practical view on substitutions, 7/2016

[373] Valdés, F., Güting, R.H.:
Index-supported Pattern Matching on Tuples of Time-dependent Values, 7/2016

[374] Sebastian Reil, Andreas Bortfeldt, Lars Mönch:
Heuristics for vehicle routing problems with backhauls, time windows, and 3D loading constraints, 10/2016

[375] Ralf Hartmut Güting and Thomas Behr:
Distributed Query Processing in Secondo, 12/2016

[376] Marija Kulaš:
A term matching algorithm and substitution generality, 11/2017

[377] Jan Kristof Nidzwetzki, Ralf Hartmut Güting:
BBoxDB - A Distributed and Highly Available Key-Bounding-Box-Value Store, 5/2018

[378] Marija Kulaš:
On separation, conservation and unification, 06/2019

[379] Fynn Terhar, Christian Icking:
A New Model for Hard Braking Vehicles and Collision Avoiding Trajectories, 06/2019