MIDET: A Method for Indexing Traffic Events

Mariana M. G. Duarte[1], **Marcos V. Pontarolo**[1], **Rebeca Schroeder**[2], **Carmem S. Hara**[1]

[1] Universidade Federal do Paraná, Curitiba, Brazil

[2]Universidade do Estado de Santa Catarina, Joinville, Brazil

`{mmgduarte, mvp19, carmem}@inf.ufpr.br, rebeca.schroeder@udesc.br`

Abstract. *Traffic events announcements such as jams and road closures are continuously reported by mobile and Web applications. This collection of spatio-temporal data is an important source of information for urban planning, and can be used to orchestrate a number of actions to improve the mobility, such as traffic control, traffic lights synchronization and preventive maintenance. Such analysis usually involves computation of spatial relationships among data, and may involve location of landmarks, roads and different types of events. In this paper, we propose a* **M***ethod for* **I***ndexing* **T***raffic* **E***vents (MIDET) for querying spatio-temporal data, whose location can be represented as a point or collection of points. MIDET is based on a fixed-grid space-oriented partitioning. In order to tackle the data skew, each grid cell is associated with a set of blocks containing event records. Moreover, a bitmap index is used for filtering out blocks without retrieving the actual data. MIDET provides the following benefits: adoption of a simple bulk loading process to manage dynamic insertion streams, and in-memory spatial joins. We conducted an experimental study using real data obtained from Waze. MIDET's query performance was compared with Postgis, which adopts an R-tree index structure.*

1. Introduction

Traffic events announcements have become an integral part of our daily lives. Road closures, jams, and active construction and maintenance are examples of traffic events that are reported in real time by applications such as Waze, and by Web portals such as the Maryland Department of Transportation[1].

This collection of spatio-temporal data is an important source of information for urban planning, and can be used to detect patterns of events in order to plan a number of actions to improve the mobility, such as traffic control, traffic lights synchronization and preventive maintenance. Such analysis usually involves computation of spatial relationships among data, and may involve location of landmarks, roads and different types of events. That is, the analysis requires the ability to combine more than one type of spatial data that may have a temporal element. For example, one may be interested in determining traffic alerts and jams that occurred in the same region and time, or the city district with the highest number of alerts in the previous month. These operations on spatial data are usually computationally expensive.

There are a number of previous works that focus on improving the performance of queries involving spatial relationships (please refer to [Chaudhry et al. 2020] and

[1]`https://chart.maryland.gov/incidents/index.php`

[Mahmood et al. 2019] for recent surveys on the subject). One of the approaches for optimizing queries is indexing. Spatial indexes usually involve a two-step process: filter and refine. The filter step reduces the search space to the spatial objects that can potentially be part of the query result. It is usually based on an approximation of the object geometry, such as the minimum bound rectangle (MBR). In the refine step objects in the candidate set are evaluated to check if they satisfy the query spatial predicate. Spatial indices can be of two classes: data-oriented and space-oriented. R-tree [Guttman 1984] is an example of a data-oriented partitioning, in which the values of the spatial attribute of objects in the dataset determine how they are stored and clustered. Space-oriented partitioning imposes a decomposition of the space in regular-sized partitions (also called tiles and grid cells - GCs). Objects that overlap the same GC are stored in blocks. The grid file [Nievergelt et al. 1984], filter trees [Sevcik and Koudas 1996], PBSM[Patel and DeWitt 1996], Paradise [Patel et al. 1997] and SATO [Vo et al. 2014] are examples of data structures and systems that adopt space-oriented partitioning.

In this paper, we are interested in querying spatio-temporal data involving traffic events. Some traffic events such as a road accident can be represented as a spatio-temporal discrete point, while others, such as jams, can be modeled as a sequence of spatio-temporal points. In fact, the location of the majority of traffic events can be represented either as a point or a collection of points. This is the case of Waze, which only uses these two types of spatial data to store all types of traffic events. Thus, in this paper our focus is on spatio-temporal join queries on events represented with these two types of vector data.

We propose a **M**ethod for **I**ndexing **T**raffic **E**vents (MIDET), which is based on a fixed-grid space-oriented partitioning index. Although in this paper we present MIDET as a method for querying historical traffic events, it has been designed to be fairly simple to be extended for dynamic updates with a stream of input data. In fact, this is the motivation for choosing a space-oriented instead of a data-oriented partitioning. The latter usually benefit from sorting the input data in order to generate a balanced index structure. This is not possible when events are continuously reported.

Two problems usually associated with space-partitioning techniques are: boundary objects, which results in data replication, and data skew, that is, a non-uniform distribution of objects among GCs. MIDET tackles them as follows. First, collections of points that overlap more than one GC are split into subsequences for each GC, while keeping the event identification and timestamp. Second, each GC is associated with a set of fixed-sized blocks (that correspond to disk pages) containing event records. A file is created for each time interval (configured by the user), and associated with a bitmap index. The bitmap vector contains one entry for each GC with a value 1 if there exists at least one event in the GC in that time interval and 0, otherwise. The bitmap is used by MIDET in the filter-step using fast bitwise operations in order to avoid the need to retrieve the actual data. Bitmap indexes are especially appropriate in this context, since we are considering an insert-only application. It is a well-known limitation that bitmaps are not suitable for frequently updated datasets.

Besides the approach for overcoming the data skew problem of space-oriented partitioning, MIDET presents two additional benefits: 1) a simple bulk loading process can be adopted as long as it is possible to allocate one block for each cell of the grid in memory; 2) spatial join operations can be executed in-memory as long as the number of

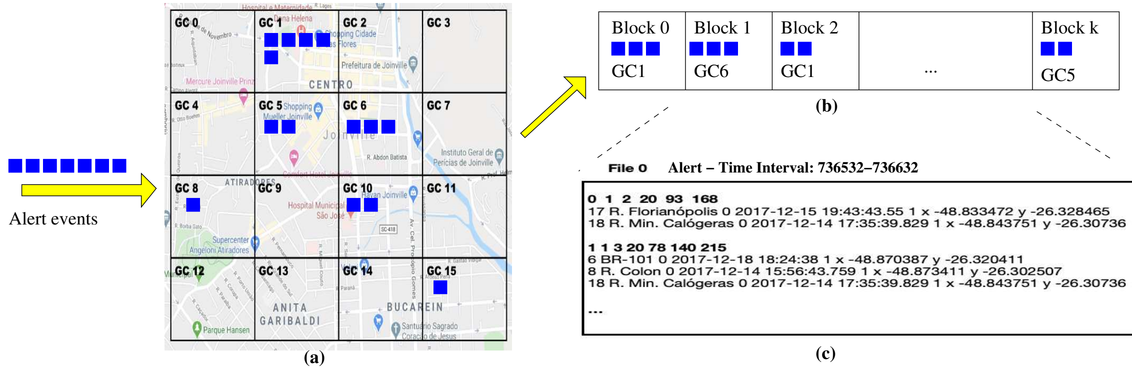


Figure 1. MIDET file organization

blocks in each GC and each time interval do not exceed the memory capacity. We ran experiments using real data obtained from Waze, containing events from Joinville, a city in the south of Brazil. Query performance is compared with Postgis with an R-tree index on the spatial attribute.

The rest of the paper is structured as follows. MIDET definitions as well as the bulk loading and query processing are described in Section 2. An experimental study is the subject of Section 3. Section 4 presents related work. We conclude in Section 5 by outlining some future work.

2. MIDET - A Method for Indexing Traffic Data

This section introduces MIDET, a ***M****ethod for* ***I****ndexing* ***T****raffic* ***E****vents*. It assumes the existence of several traffic event types, such as jams, alerts and irregularities. The method is based on two basic principles: definition of equal length time intervals and, for each interval, applies a spatial fixed grid partitioning of the events. The motivation of the approach is to optimize spatio-temporal join queries. That is, queries that combine two datasets based on their spatial and temporal attributes.

The sizes of time intervals and the grid are given as input by the user, and coincide for all event types. The interval can be defined as days, weeks or months. The grid size is defined by the number of horizontal partitions (or rows - R) and vertical partitions (or columns - C). Then, given the MBR of the area of interest, defined by its upper left coordinate *(latUL, longUL)* and lower right coordinate *(latLR, longLR)*, the vertical ($SizeLat$) and horizontal ($SizeLong$) size of each grid cell (GC) is given by $(latLR - latUL)/R$ and $(longLR - longUL)/C$, respectively. An example of a 4x4 grid on the city of Joinville in Brazil, is shown in Figure 1(a).

In MIDET, traffic events that occur within each GC are stored together. Each traffic event has a spatial attribute, which may be a point or a collection of points defined by their coordinate. In order to identify the GC that contains the point location *(lat, long)*, we determine its row and column as follows:

$$row = \left\lfloor \frac{lat - latUL}{SizeLat} \right\rfloor column = \left\lfloor \frac{long - longUL}{SizeLong} \right\rfloor$$

For traffic events with the spatial attribute defined by a collection of points (such as

traffic jams and irregularities), MIDET partitions the collection into subsets that overlap a single GC. More specifically, it analyzes whether two sequential points in the collection, p_n and p_{n+1}, are located in different GCs. If this is the case, it splits the set of points, creating two event records (with the same event ID), r_1 and r_2, where r_1 contains the starting point up to p_n, plus an additional point referring to the intersection point of the GC border and the line segment formed by p_n and p_{n+1}. This same point is inserted into r_2, along with the points starting from p_{n+1}.

MIDET adopts a mixed storage model based on files and a relational database. Event records are stored in files and the database maintains information to speed up accesses to the files.

2.1. File Organization

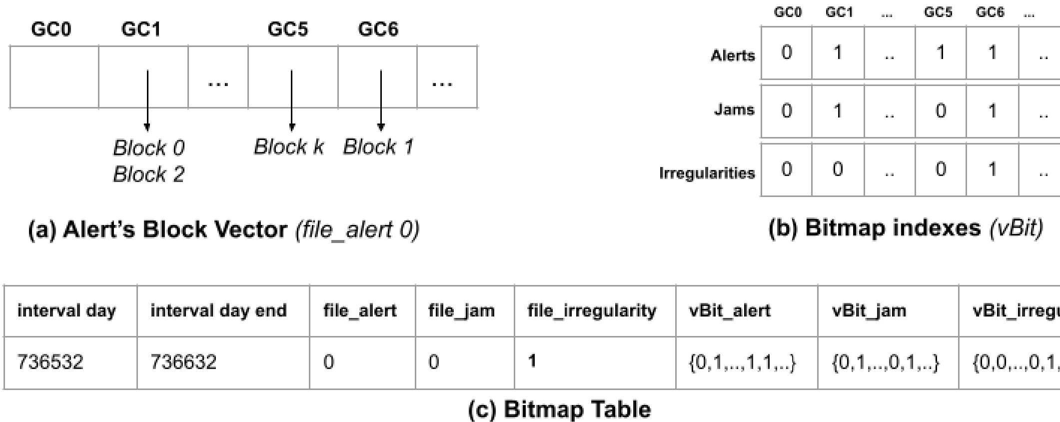
In MIDET, a file is created for each time interval and event type. Each file is composed of a set of blocks, which in turn contains a set of records. *Blocks* are transfer units between the external storage and the main memory and contain records of events that occurred in the same GC. When the number of records in a GC exceeds the predefined block size, a new block is created. By the end of each time interval, the remaining free space in each block is not filled. In summary, each time interval and event type has a corresponding file, and in this file there exists a set of blocks for each GC. Figure 1(b) shows an example of this organization.

Since event records may have variable length, each block has a header with information to access its records. Figure 1(c) illustrates the file organization and the structure of a *header*. It is composed of: block ID, referenced GC ID, number of records in the block and the address of each record within the block. The *header's* size is defined by the amount of *slots*, that is, the maximum number of records that a block can contain. In Figure 1(c), the first block has id `0`, GC id `1`, the number of records in the block `2`, the address of each record `20` and `93`, and the block first free address `168`.

2.2. Indexing

In order to optimize the retrieval of blocks that contain events of the same GC, we propose a structure inspired on an inverted index, mapping GCs to blocks, as illustrated in Figure 2(a). This structure helps the execution of range queries, that is, queries to find events that occurred in a given area of interest. However, it does not take into consideration the temporal dimension. Moreover, MIDET proposes a bitmap index to optimize spatial join queries. More specifically, each time interval and event type is associate with a bitmap vector (*vBit*), indexed by GCs, so that *vBit [n] = 1* if the file contains records of events that occurred in GC_n, and *vBit [n] = 0*, otherwise. Figure 2(b) shows 3 bitmap indexes, each one associated with a file. For the traffic jam's bitmap, the index shows that the associated file contains records that occurred in GCs 1 and 6, but no records in GCs 0 and 5. By storing these bitmaps, it is possible to compute a fast bitwise operation between the events to filter the GCs that contain candidates to be joined.

All bitmap indexes associated with the files are stored in a relational database in a table called Bitmap. Each row of the Bitmap Table corresponds to a time interval, with the conversion of years and months to days. Bitmap table's attributes are: the beginning and end of the interval in days, the bitmap indexes for each type of traffic event, and the



(a) Alert's Block Vector *(file_alert 0)*

(b) Bitmap indexes *(vBit)*

interval day	interval day end	file_alert	file_jam	file_irregularity	vBit_alert	vBit_jam	vBit_irregularity
736532	736632	0	0	1	{0,1,..,1,1,..}	{0,1,..,0,1,..}	{0,0,..,0,1,..}

(c) Bitmap Table

Figure 2. MIDET Indexing

file counter. For example, according to the table line illustrated in Figure 2(c), event logs for the period between `interval day` and `interval day end` are stored in files named `Alert0`, `Jam0` and `Irregularity1`. That is, it is the first file created for alerts, the first file for jams, and the second file for irregularities. The table also contains the bitmaps illustrated in Figure 2(b).

A block vector table is created for each type of traffic event. This table associates each GC with blocks that compose the files, and corresponds to the inverted index illustrated in Figure 2(a). The table has three information: the file counter, the GC identification, and the block address in this file.

2.3. Bulk Loading

MIDET's process for generating the database receives as input the following parameters: the area of interest with the upper left and lower right coordinates, the maximum block size in bytes, the number of *slots'* in each block, the grid size, given by the number of rows and columns, and the time interval in days.

For simplicity, in the following, we consider the creation of the database for a single event type. At the beginning of each time interval, a page of the size of a block is allocated in memory for each GC. Moreover, a bit vector indexed with the GCs is created with all bits set to zero, and a new file is created. We assume that the sequence of events is reported in chronological order. Then, given an event, MIDET first determines the GC of its location, and if necessary, splits the collection of points, according to the process detailed before. At each record insertion, it is checked whether the sum of the record's size and the used space in the page is greater than the maximum predefined block size. If so, the content of the page is written to the file on disk and a new line is created in the *Block Vector Table* with a reference to file, GC, and block address. Then, an empty page is created to store the record in memory. On the other hand, the record is appended to the existing page if there is enough space left. Moreover, the bit corresponding the record's GC is set to one in the bit vector.

At the end of a time interval, all pages are written to file on disk, the file is closed and a new line in the bitmap table is created with information of the files and bitmaps created during the process. Next, we will detail how spatial join queries are processed on

MIDET.

2.4. Query Processing

Query processing in MIDET is illustrated in Figure 3. In the example, a spatial-join between jams and alerts is represented. First, the file name of each event and their bitmap vectors are obtained from the *BitmapTable*, considering a time interval as a query predicate. Second, GCs with value 1 in both bitmap vectors are selected using a bitwise operation. It means that we apply a pruning strategy based on the existence of records for both events in that GC. Besides, we avoid a full scan on event files by retrieving the block addresses on files which contain records for that specific GC. They are obtained from the *Block Vector Table*. Then, the respective blocks are read to main-memory, and the information in their *headers* are used to access each record in the block. Finally, jam and alert records are joined and the query predicates are applied.

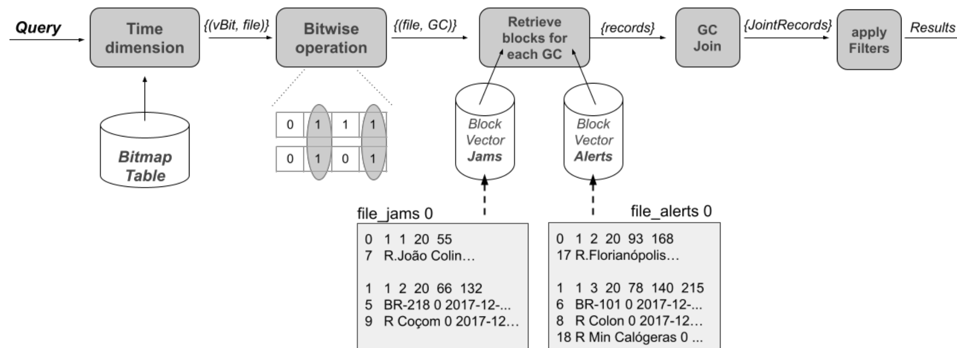


Figure 3. Example of Query Processing in MIDET

We assume that join operations as well as filters can be executed in-memory as long as the number of blocks in each GC does not exceed the memory capacity. Otherwise, additional reads of block vectors may be necessary to avoid disk accesses. Such issue will be addressed as a future optimization of the algorithm, as well as the filter pushdown.

3. Experimental Study

This section describes MIDET's implementation and a experimental study. MIDET has been implemented in C, and compiled using gcc 11.0.0. The relational database management system adopted was Postgres version 12.5 with the Postgis extension. For the connection between C and Postgres, the libpq[8] library was used. For creation, storage and processing bitmaps, we used the roaring bitmap ([Lemire et al. 2018]). It has been chosen based on a study [Wang et al. 2017] that concludes that it is a superior alternative compared to other compressed bitmaps. We used CRoaring[6], which is its implementation in C and the extension pg_roaringbitmap[7] for Postgres. The entire implementation code can be found at gitlab[5].

The experimental study was conducted on a real Waze dataset, which was obtained from the project *Smart Mobility* of Joinville, a city in the south of Brazil. Access to

[8]https://www.postgresql.org/docs/9.5/libpq.html
[6]https://github.com/RoaringBitmap/CRoaring
[7]https://github.com/ChenHuajun/pg_roaringbitmap
[5]https://gitlab.com/mmgd/midm

the data was granted to the State University of Santa Catarina, partner of this work. The source base *Waze* has 13 Gigabytes (GB), containing data from September 2017 to September 2018, with three types of events: jams, alerts, and irregularities. We generated the MIDET database with a subset of attributes from the original dataset: event ID, street name, timestamp, and geometry. For comparison purposes, a relational database with the same set of attributes were created, with one relation for each event. These relations were indexed with an R-tree over the geometry attribute. We denote this database as Waze.

In order to run experiments on different dataset sizes, we created 5 Waze and MIDET databases, (*B20*), (*B30*), (*B40*), (*B50*) and (*B100*), that correspond to 20%, 30%, 40%, 50% and 100% of the original dataset. To build these databases, the *central* region of the city of Joinville was chosen as the initial area of interest for the B20 database, as it contains a greater concentration of traffic events, and thus contains partitions with higher density and others with no events. The other databases are extended with events occurred towards the city border. Thus, database *B30* contains all records in *B20*, with additional 10% of events. The *B100* database contains all records of the original dataset. Table 1 shows the number of records in each database, as well as the number of GCs that cover the events in the database.

Base	Percentage	#alerts	#jams	#irregularities	#GCs
B20	20%	1024311	587816	22063	81
B30	30%	1536466	887724	33095	120
B40	40%	2048622	1183632	44126	161
B50	50%	2560777	1479540	55158	198
B100	100%	5121554	2959080	110316	400

Table 1. Number of records for each event type

All experiments were executed on a desktop running Linux Mint 20 Ulyana operating system, with 4.00GHz Intel Core i7-4790K, and 16GB of RAM.

The MIDET input parameters were set as follows. The grid was created with 20 rows (L) and 20 columns (C) over an area of interest of 625 km^2 delimited by a rectangle with the left upper coordinate at (-26,-49.3) and the right lower coordinate at (-26.5,-48.83). Each grid cell has 1.56 km^2. The time interval was set to 365 days, the maximum block size was defined as 8 *KiloBytes*(KB) to coincide with the page size used by Postgres. Moreover, the file header was defined with 180 slots; that is, each block can contain at most 180 event records.

3.1. Database creation

We first report on the databases loading time. The time to create the MIDET and Waze databases are presented in Figure 4. The time reported for MIDET consists of the time to create the files for each event time, along with the bitmap and block vector tables in Postgres. For Waze, the creation time consists of the creation of a table for each event time, indexed and clustered by their geometry attribute. The results show that MIDET performs better than Waze, especially because it explores the fact that traffic events are reported in chronological order and record clustering is defined by the GC of the event location, which is easily computed. Moreover, the file organization minimizes disk writes to two conditions: either when the block of the GC is full or when a time interval expires.

Base	MIDET	Waze
B20	20s425ms	43s149ms
B30	32s734ms	57s958ms
B40	43s962ms	1min21s142ms
B50	56s506ms	1min51s934ms
B100	1m50s055ms	6m15s998ms

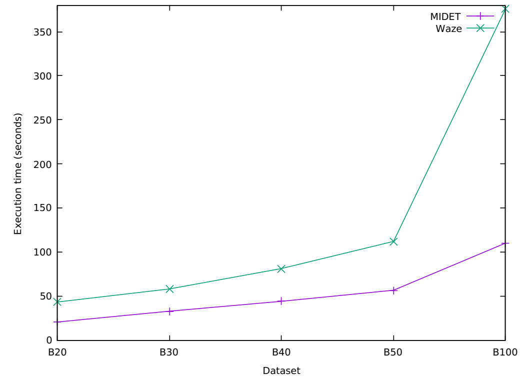


Figure 4. Time to create MIDET and relational Waze databases

The volume of the resulting databases are presented in Table 2. For each type of event, the table shows for MIDET: the total number of blocks in all files, the total size in MBytes of the files plus the bitmap and vector blocks tables. For Waze, the volume comprises the size of the relational table for the event. The table shows that for alerts, the volume of the MIDET database is roughly 22% smaller than Waze, while for jams and irregularities it is 13% and 27% greater than Waze. This is because the geometry of alerts is a point while the other two are collection of points. Thus, whenever the set of points overlaps more than one GC the record is duplicated in MIDET with the subset of points inside each GC.

Base	Alerts			Jams			Irregularities		
	MIDET		Waze	MIDET		Waze	MIDET		Waze
	#blocks	total (MB)	total (MB)	#blocks	total (MB)	total (MB)	#blocks	total (MB)	total (MB)
B20	7503	60.07	77	10473	77.30	68	569	4.07	3.27
B30	11905	95.46	123	15228	112.80	100	762	5.48	4.34
B40	15543	124.72	160	20866	154.64	138	1092	7.79	6.09
B50	19427	155.99	300	26067	193.17	172	1320	9.40	7.42
B100	38771	313.15	399	52524	391.55	345	2623	18.92	15.00

Table 2. Database Size

3.2. Query Processing

In order to analyze the processing time of spatial joins, we consider the following query: “Which streets had a traffic jam and an alert at the same point after the first day of September, 2017?" The corresponding query in SQL is presented in Figure 5.

```
SELECT i.street
FROM jams i, alerts j
WHERE i.pub_utc_date > '2017-09-01 00:00:00'
  and j.pub_utc_date > '2017-09-01 00:00:00'
  and i.street=j.street
  and ST_Intersects(i.geom,j.geom)
```

Figure 5. Query with a spatial join

The processing times of this query is shown in Figure 6. It can be noticed that MIDET performs better than Waze, but the gain reduces for larger datasets. For B20 MIDET reduces the execution time by 42%, while for B100 the reduction is of 19%. We conjecture that for some GCs the number of blocks did not fit in memory, which led to a higher execution time for MIDET. These results are consistent with the ones reported previously [Pavlovic et al. 2016], which states that space-oriented partitioning performs better than data-oriented partitioning for joining datasets with similar density. This in fact is a characteristic of traffic events. That is, alerts such as car accidents, tend to cause jams, and thus the spatio-temporal similarities among different types of events are usually high. Moreover, the difference in processing times is due to our strategy of filtering GCs containing candidates using a fast bitwise operation and restricting the spatial operator to pairs of records that occurred in the same GC. These results show that MIDET can reduce the query processing times of spatio-temporal join queries involving traffic events. We intend to consider other types of queries in the future.

Base	MIDET	Waze
B20	12s929ms	22s311ms
B30	18s623ms	29s688ms
B40	38s183ms	1min3s
B50	57s653ms	1min21s
B100	3min13s147ms	3min58s

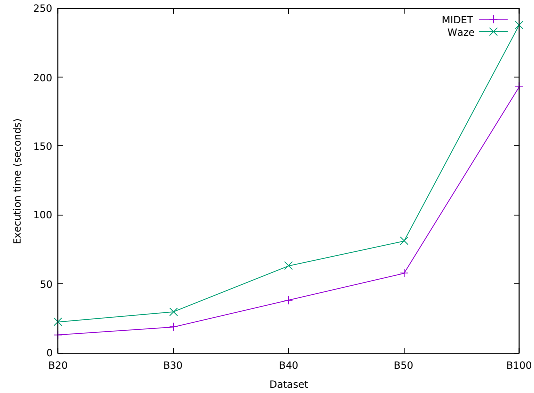


Figure 6. Query processing time

4. Related Work

According to [Aji et al. 2015], there are six most commonly used approaches for spatial partitioning: fixed grid, binary split, strip, boundary optimized strip, Hilbert curve, and sort-tile-recursive. Moreover, it analyzes the quality of the generated partitions on two datasets, as well as their join spatial performance on a MapReduce cluster, and the cost of partitioning the dataset. Given the simplicity of the fixed grid partitioning, its cost to generate the partitions is several orders of magnitude lower than the others. This indicates its potential as a technique to store streams of spatio-temporal data and motivated us to adopt it for MIDET. Most of the existing partitioning techniques have not been proposed as a storage strategy, but for the filter step of spatial joins. That is, when joining two datasets by their spatial attribute, both datasets are initially divided into subsets that overlap the same geographic cell (GC). Fixed grid partitioning has been adopted for the filter step of disk-based joins by PBSM [Patel and DeWitt 1996], and more recently explored to enable in-memory execution of the refine step for each GC [Tsitsigkos et al. 2019, Nobari et al. 2017] or to execute it in parallel [Vo et al. 2014, Eldawy and Mokbel 2015, Yu et al. 2015]. MIDET is similar to PBSM with respect to the steps to execute joins, but differs on its storage model.

PBSM file structure is similar to the grid file [Nievergelt et al. 1984], which is one of the first works that focused on the storage of grid partitioned data. In fact, the grid file is one of the several point access methods detailed in a multidimensional access method survey [Gaede and Günther 1998]. The grid file proposes a many-to-one relationship between grid cells and disk pages. That is, the space is partitioned into cells and whenever the page capacity is exceeded it causes a page split. Moreover, at each split, the entire grid doubles the number of cells. However, the grid cells that do not caused the split keep their references to the same disk page. MIDET is similar to grid files, but its relationship from grid cells to disk pages is one-to-many. More specifically, MIDET keeps a fixed-grid partition, and whenever the page capacity is exceeded, an additional page (that we call block) for the same partition is allocated. Some works built on grid files by proposing tree-based structures to associate grid cells to disk pages. GE-tree [Shin et al. 2019] adopts a quad-tree, while MSI [Al-Badarneh et al. 2013] adopts an R-tree. In contrast to these works, MIDET adopts a bitmap vector to represent the existence of data in each cell in order to handle unbalanced distribution of the data, and stores the association of cell to files and pages in a relational database.

Similar to MIDET, there are some works that consider bitmap-based indexing, either associated with a tree structure [Neto et al. 2013, Siqueira et al. 2012], or associated with space partitioning, such as BPSJ [Shohdy et al. 2015]. BPSJ proposes a parallel spatial join algorithm that scans the spatial space to find suitable points to split the space in order to generate balanced partitions, followed by an in-memory bitmap-based join. However, the bitmap vectors are of the size of the dataset. MIDET's bitmap vectors are much smaller, of the size of the grid. [Antoine et al. 2011] is another work that adopts a bitmap index. However, as opposed to MIDET, it considers a hierarchical partition of the space in order to avoid objects to cross cell borders, and associates objects to the lowest partition that fully contains the object. A file and bitmap index is associated to each partition and level. In contrast, MIDET associates a file to each time interval and considers a single level grid structure.

Some existing works propose methods that target specifically mobility data management. Among them we mention two: STIG tree [Doraiswamy et al. 2016] and TQ index [Imawan et al. 2015]. Both propose special structures for indexing mobility data that are *not* based on a grid. STIG tree [Doraiswamy et al. 2016] proposes a KD-tree for indexing spatio-temporal data with sets of events on the leaves that can be processed in parallel on GPUs. TQ index [Imawan et al. 2015] was proposed to predict traffic jams based on spatio-temporal traffic data. The model is optimized for analytical queries and maintains two main components: a location index and a time index, which are based on hash tables. Although MIDET considers the same type of data as STIG tree and TQ Index, it follows a different approach: a fixed grid partitioning with a bitmap-based indexing and a file structure with a set of blocks associated to grid cells.

5. Conclusion

This paper proposes a method for storing and indexing traffic events data called MIDET. It explores the spatio-temporal similarities of different types of events to develop a method that combines the two-step spatial-oriented partitioning introduced by PBSM with a file organization similar to grid files. Moreover, it uses a bitmap indexing over grid cells to help filtering out partitions that cannot contribute in the production of results for spatial join

queries. MIDET has been implemented, storing event records in space-partitioned block files, and building indexes in a Postgres relational database with the pg_roaringbitmap extension. We have conducted an experimental study on real Waze data. It showed that MIDET presents better performance for generating the database and for executing spatial joins compared to Postgis with an R-tree index over the events geometry attribute.

The storage organization proposed by MIDET has been developed as the basis for some of the future work we intend to pursue. We aimed at a structure that can be fairly easily adapted to: 1) dynamic insertion of streaming data, 2) parallel processing of spatial joins; and 3) dynamically be able to modify the length of time intervals to guarantee in-memory joins. Future works also include a method for automatically detect best GC sizes and change the temporal configuration whenever joins are not possible to be executed in-memory. Moreover, we intend to conduct a more comprehensive experimental study, including a comparison with alternative index structures, and using synthetic datasets or benchmarks in order to analyze in detail the impact of data density and distribution.

References

Aji, A., Vo, H., and Wang, F. (2015). Effective spatial data partitioning for scalable query processing. *CoRR*, abs/1509.00910.

Al-Badarneh, A. F., Al-Alaj, A. S., and Mahafzah, B. A. (2013). Multi small index (MSI): A spatial indexing structure. *Journal of Information Science*, 39(5):643–660.

Antoine, E., Ramamohanarao, K., Shao, J., and Zhang, R. (2011). Accelerating spatial join operations using bit-indices. In *Proc. of the 22nd Australasian Database Conference*, volume 115, page 123–132.

Chaudhry, N., Yousaf, M. M., and Khan, M. T. (2020). Indexing of real time geospatial data by IoT enabled devices: Opportunities, challenges and design considerations. *Journal of Ambient Intelligence and Smart Environments*, 12:281–312.

Doraiswamy, H., Vo, H. T., Silva, C. T., and Freire, J. (2016). A GPU-Based Index to Support Interactive Spatio-Temporal Queries over Historical Data. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, Helsinki, Finland.

Eldawy, A. and Mokbel, M. F. (2015). Spatialhadoop: A mapreduce framework for spatial data. In *Proc. of the 31st International Conference on Data Engineering*, pages 1352–1363.

Gaede, V. and Günther, O. (1998). Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231.

Guttman, A. (1984). R-trees: a dynamic index structure for spatial searching. *ACM Sigmod Record*, 14(2):47–57.

Imawan, A., Putri, F., and Kwon, J. (2015). TiQ: A Timeline query processing system over Road Traffic Data. In *2015 IEEE International Conference on Smart City*, Chengdu, China.

Lemire, D., Ssi-Yan-Kai, G., and Kaser, O. (2018). Consistently faster and smaller compressed bitmaps with roaring. *Software: Practice and Experience*, 46:1547–1569.

Mahmood, A. R., Punni, S., and Aref, W. G. (2019). Spatio-temporal access methods: a survey (2010 - 2017). *Geoinformatica*, 23:1–36.

Neto, C. J., Ciferri, R. R., and Santos, M. T. P. (2013). HSTB-index: A hierarchical spatio-temporal bitmap indexing technique. In *SBBD - Workshop de Teses e Dissertações*.

Nievergelt, J., Hinterberger, H., and Sevcik, K. C. (1984). The grid file: An adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems*, 9(1):38–71.

Nobari, S., Qu, Q., and Jensen, C. S. (2017). In-memory spatial join: The data matters! In *Proc. of the 20th International Conference on Extending Database Technology (EDBT)*, pages 462–465.

Patel, J. M. and DeWitt, D. J. (1996). Partition based spatial–merge join. In *Proc. of the 1996 ACM SIGMOD international conference on Management of data*, pages 259–270.

Patel, J. M., Yu, J., Kabra, N., Tufte, K. A., Nag, B., Burger, J., Hall, N. E., Ramasamy, K., Lueder, R., Ellmann, C. J., Kupsch, J., Guo, S., Larson, J. G., Witt, D. J. D., and Naughton, J. F. (1997). Building a scaleable geo-spatial dbms: technology, implementation, and evaluation. In *Proc. of the 1997 ACM SIGMOD international conference on Management of data*, page 336–347.

Pavlovic, M., Heinis, T., Tauheed, F., Karras, P., and Ailamaki, A. (2016). Transformers: Robust spatial joins on non-uniform data distributions. In *Proc. of the 32nd International Conference on Data Engineering (ICDE)*, pages 673–684.

Sevcik, K. C. and Koudas, N. (1996). Filter trees for managing spatial data over a range of size granularities. In *Proc. of the 22nd International Conference on Very Large Data Bases (VLDB)*, page 16–27.

Shin, J., Mahmood, A., and Aref, W. (2019). An investigation of grid-enabled tree indexes for spatial query processing. In *Proceedings of the 27th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 169–178.

Shohdy, S., Su, Y., and Agrawal, G. (2015). Load balancing and accelerating parallel spatial join operations using bitmap indexing. In *Proc of the IEEE 22nd International Conference on High Performance Computing (HiPC)*, pages 396–405.

Siqueira, T. L. L., de Aguiar Ciferri, C. D., Times, V. C., and Ciferri, R. R. (2012). The SB-index and the HSB-index: efficient indices for spatial data warehouses. *Geoinformatica*, 16(1):165–205.

Tsitsigkos, D., Bouros, P., Mamoulis, N., and Terrovitis, M. (2019). Parallel in-memory evaluation of spatial joins. In *Proc. of the 27th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 516–519.

Vo, H., Aji, A., and Wang, F. (2014). SATO: A spatial data partitioning framework for scalable query processing. In *Proc. of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, page 545–548.

Wang, J., Lin, Y. C., and S., S. (2017). An experimental study of bitmap compression vs. inverted list compression. In *Proceedings of the 2017 ACM SIGMOD International Conference on Management of Data*, New York, NY.

Yu, J., Wu, J., and Sarwat, M. (2015). Geospark: A cluster computing framework for processing large-scale spatial data. In *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*, New York, NY, USA. Association for Computing Machinery.