A Database for Repetitive, Unpredictable Moving Objects

Paul Werstein and J. R. McDonald

Department of Computer Science
University of Otago
Dunedin, New Zealand
werstein@cs.otago.ac.nz, jrm@cs.otago.ac.nz

Abstract. Recently much research has been done in applying database technology to tracking moving objects. Most techniques assume a predictable linear motion. We present a new application for moving object databases, characterized by repetitive, unpredictable motion with very high data rates. In particular, the domain of athletic and auto races is presented. We found existing moving object methods do not adequately address this application area. A data model is presented for efficiently storing the data. A spatiotemporal index is developed for fast retrieval of data. We give a set of queries likely to be of interest in this application domain. A study is presented showing our implementation has better performance than those based on relational DBMSs.

1 Introduction

In the last few years, much research has been done in applying database technology to tracking moving objects. The techniques developed to date assume a predictable, linear motion so an object can be represented as a position, time at that position, and some motion vector to give future locations.

This research looks at a class of applications which are characterised by repetitive, unpredictable motion and high data rates. Examples include sporting events such as athletic track and field races, bicycle races and car races. This class of applications cannot be represented using existing techniques.

The motion is repetitive since it involves many laps around a closed course. Each object (person, bicycle, car) covers approximately the same space which is the designated race course. This causes problems for indexing techniques using minimum bounding boxes since they expand to cover the entire data space. Problems also occur for techniques which divide the data space such as quadtrees since objects will appear in many of the rectangles.

The motion is unpredictable in the sense that an equation or set of equations cannot be used to describe the motion of the objects. It may be possible to describe mathematically an ideal path. However external, unpredictable factors cause the motion to vary frequently from the ideal path. In track and field races, factors include other athletes in the path, variation due to wind, and slips. In auto races, factors include other cars in the path, debris or water on the track, unevenness of the track, drivers' skills, momentary loss of control and accidents.

This application class requires a high data rate to achieve the desired accuracy. We use Formula One car races as our example. The race cars are very close to each other and travelling at high speeds. At a top speed of approximately 360 kph, they travel 100 meters per second. We chose to have a resolution of 100 mm which means 1000 points per second are required. Fortunately, a race is relatively short and has a small number of objects (22 cars and about 90 minutes in our test set) which limits the amount of data to be stored. However it still amounts to over 7 GB of data.

The contribution of this research is a method of storing the locations of moving objects with a high data rate and indexing them for rapid retrieval. We found that conventional relational databases cannot handle adequately the data storage and retrieval needed to record a race and to allow display of the data in real time.

The next section positions this work in relation to other research. Section 3 describes how data is stored and indexed. Section 4 describes the performance of our database, and Section 5 has concluding remarks.

2 Related Work

Much of the current research on moving objects uses scenarios such as a car moving on a highway [9, 2]. These types of applications are inherently different from our research scenario in the following ways:

- The requirements for positional accuracy are much higher in our application. For example, if a driver queries for hotels within 5 km, inaccuracy of even a few hundred meters is of little concern. However, for race cars traveling less than a meter apart, high accuracy is required.
- Speed changes are rather dramatic in our application. Most current research assumes fairly constant speed so the motion can be expressed as a linear equation [6]. Car races are characterised by continual speed changes. Formula One cars are capable of braking up to five G's and acceleration exceeding one G. They frequently slow for turns to about 50 kph and accelerate quickly to about 360 kph, so their speed changes very rapidly.
- It is difficult if not impossible to model a car's motion by an equation or set of equations. Many authors use a linear equation to define motion between observations [9, 6, 10]. For reasons mentioned previously, it is not possible to describe the actual path of a race car by a set of equations. Another application of our techniques is to track animals or hikers in a wilderness area. There are few if any constraints to their motion which makes mathematical description impossible.

Several data models have been proposed for representing moving objects in a database. The MOST data model, proposed in [9], incorporates dynamic attributes which are attributes whose values are a function of time. A dynamic attribute is represented by three sub-attributes, *A.value*, *A.updatetime*, and *A.function*. *A.function* is a function of time that has a value 0 at time = 0 and describes the manner in which the attribute value changes over time. The value of the attribute at *A.updatetime* is *A.value*. Thus at any future time, t > *A.updatetime*, the value of the attribute is *A.value* + *A.function(t)*.

Wolfson et al. use a data model which stores an object's start time, starting location, and a velocity vector [12]. Database updates are generated by the object when its location deviates more than a given threshold from the location that would be computed by the database.

In order to process queries in reasonable time, the data must be indexed. Some authors have discussed indexing present and future positions of moving objects [9, 11, 5, 8, 1]. Other authors discuss indexing the past positions of moving objects [7].

In [9], the authors propose indexing their dynamic attributes. Two alternate approaches are suggested in [11]. The first is the value-time representation space where an object is mapped to a trajectory that plots its location as a function of time. In the alternative intercept-slope representation space, the location is a function of time, $f(t) = a + vt$.

A dual transformation scheme is also used in [5] in which the trajectories of moving objects are transformed from the time-trajectory plane into a slope-intercept plane. They use an R*-tree and hB-tree to study the performance of the transform.

In [8], Saltenis et al. use a variant of the R*-tree called a TPR-tree (time parameterised tree) to index the current and future locations. While good for predicting locations in the near future, the minimum bounding boxes would grow to cover the entire track in our application.

An alternative indexing scheme for future locations is presented in [1], where constant speed motion in a straight line is assumed. Chon at al. present an index scheme for past, present, and future positions [2]. They use a space-time partitioning scheme where the data space is divided *a priori*. Their assumption is that moving objects have no more than hundreds of points. The grid is maintained in main memory.

In [7], the authors use variants of an R-tree, called STR-tree and TB-tree, to index past histories. These are variants in the sense that trajectories instead of locations are contained in the minimum bounding boxes. In our application, the trajectories are constantly changing which leads to a very large number of bounding boxes.

3 Data Storage and Indexing

Several authors have suggested building moving object databases on top of relational or object-relational databases [9, 11, 4, 3, 6, 10]. However, our experiments show that relational DBMSs fall short in two ways.

First, a relational database returns a relation as the result of a query. In this paradigm, the database server usually creates a relation for the entire result and then delivers it to the requesting client. For some queries such as replaying an entire race, the result is far too large to deliver at once. Since the result cannot fit into main memory, it is temporarily written to disk, further adding to overhead. Instead we want to locate quickly the beginning of the data and then deliver it incrementally in a time frame that is no worse than actual race time. This allows for a reasonable visualisation of the data in a graphics display.

Secondly, we found that a sample of relational DBMSs simply cannot answer the queries we posed. These queries are given in Section 4. We tried our application on a popular commercial relational database, PostgreSQL and mySQL. The load times far exceeded the sample race time. In addition, some queries failed to complete in a reasonable time.

In our data model, data is stored as (t, x, y, l, s, h, Z) where t is the time of the sample; (x-y) is the location at time t; l, s and h are the lap number, speed, and apparent heading at time t; and Z represents any other attributes that may be desired.

The data is stored on disk so that all data for each entrant is in time sequence. Thus once the start of the desired data is located, data may be read sequentially.

Considering the nature of the queries (described in Section 4), three indexes are constructed. The first index and the main contribution of this research is the spatiotemporal index. The approach we take is to partition the data space uniformly by the area covered. This is in contrast to the R-tree and its variants which use minimum bounding boxes to partition the data space. It is also in contrast to techniques such as a quadtree which recursively divides the area. Instead we divide the area into uniform subareas according to the expected data. This division is done before data is loaded although one could easily rebuild the index if another division were desired.

As an example, consider the case of an oval track. Figure 1 shows the track with a subdividing grid superimposed on it.

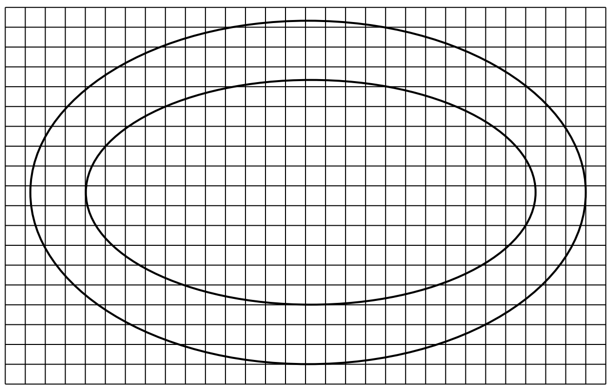

Fig. 1. An oval track with a grid superimposed

The spatiotemporal index employs two levels. At the first level is a logical X-Y matrix using sparse matrix techniques. That is, the only cells which are allocated memory or disk space are those that contain actual data (See Figure 2). We begin by allocating space for the entire distance expected to be covered in the X-axis. Cells are allocated for the Y-axis as needed. Thus the X-Y matrix only contains cells that cover area occupied at some point in time by the moving objects.

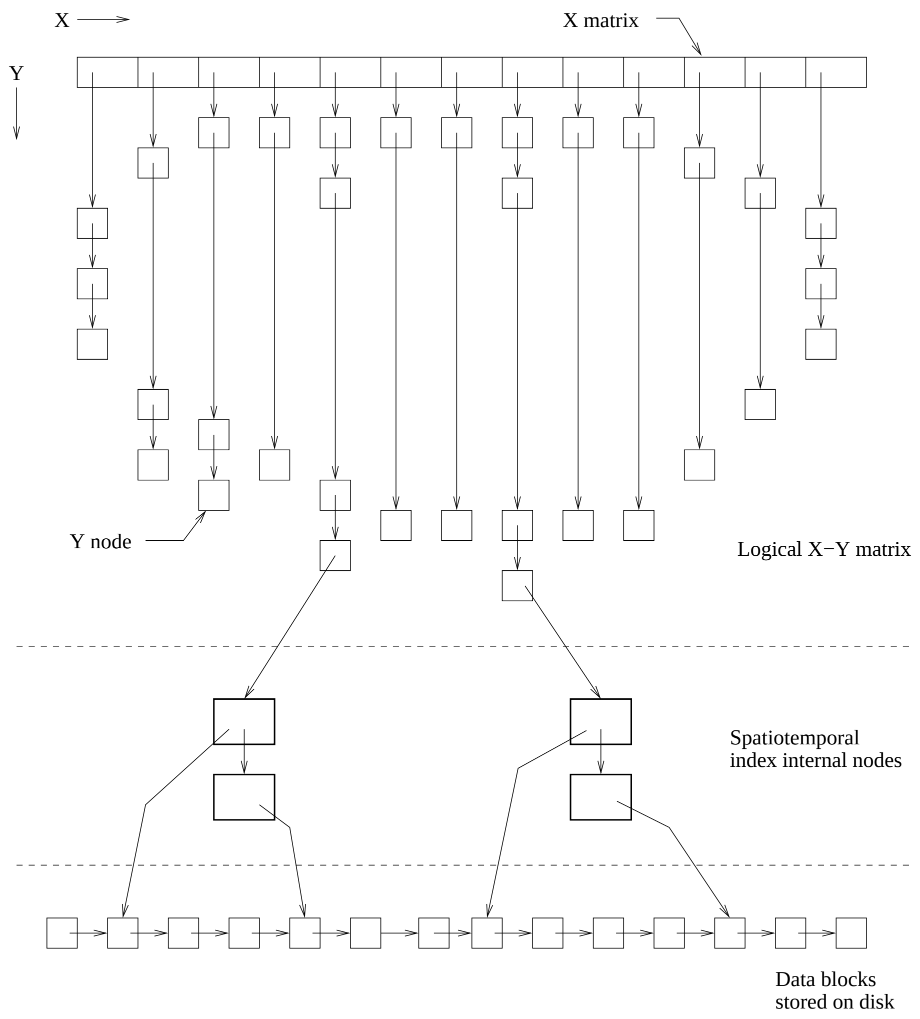
X
X matrix
Y
Y node
Logical X−Y matrix
Spatiotemporal index internal nodes
Data blocks stored on disk

Fig. 2. Spatiotemporal index data structure

In Figure 2, the logical X-Y matrix is shown above the upper dashed line. The disk blocks containing the race data are shown below the lower dashed line. The internal part of the spatiotemporal index is shown between the dashed lines.

Each X-Y matrix cell is linked to one or more cells containing the internal spatiotemporal index entries. An index entry is of the form (t, l, d) where t is the time when the object entered that space, l is the lap number the object is on when it entered, and d is the data block on disk containing the actual data. The three entries, (t, l, d), are used to align query results properly (explained below).

The second index is a variation of a B^+-tree on time. This index is a variant in the sense that each node is full rather than one-half or two-thirds full since there is no need to leave room for future insertions. In addition there is only one entry for each data block. We call this a B^{++}-tree.

Splitting is done differently. When a node becomes full, a new node at the same level is created. Entries do not need to be moved between the original node and the new node. Instead the new entry is placed into the new node and pointer is sent up the tree where the process may be repeated. A node may be written to disk once it is full since it will not be subject to future splitting.

Finally, lap numbers are indexed by a B^{++}-tree. This index allows for very fast data location when playing back a particular lap.

4 Performance

We have implemented a database engine and the spatiotemporal indexing scheme in C++ on Linux. We call the implementation RSDB. The results described were obtained on a 500 Mhz Pentium III based PC with 128 MB of memory.

We use a data load and three queries to compare RSDB to relational databases. These queries represent the data required to replay certain parts of a race similar to the so-called 'instant replays' often seen on television broadcasts of such events. Comparisons are made against a commercial DBMS, PostgreSQL and MySQL. Our licensing agreement prevent us from publishing the results of the commercial DBMS. However it is fair to say that it also was lacking sufficient performance for this application.

To evaluate the performance of the indexing, a database was loaded with synthetic data which approximates a Formula One race. The files required to store the actual data use 330 MB of disk space per object. The time index uses 1 MB of disk space (0.3% of data space) and the spatiotemporal index uses 6.6 MB (2.0% of data space). Total disk space for the example race in our implementation was 7.6 GB.

For the relational DBMSs, each car has its own relation which makes it much more efficient when retrieving data for only a few cars. For RSDB, the data for each car is stored in a separate file.

4.1 Loading

A data generating program was developed for loading the test databases. This program generated data similar to the Formula One Grand Prix at Hockhenhiem, Germany. In 1999, the race lasted 4919 seconds (1 hour 22 minutes). The simulated race lasts 5064 seconds (1 hour 24.4 minutes). When the data generation program is not actually interfacing to a database, it takes 80 seconds to run. Thus the time to generate the data is small compared to the time of the simulated race.

The time to load the data and to build indexes is shown in Table 1.

Table 1. Time to load data and build indexes. Time in hrs:min:sec

System	Load data	Build indexes
PostgreSQL	59:01:05	5:55:40
MySQL	8:41:50	33:45:50
RSDB	1:00:56	0:16:35

Normal SQL commands were used to load the relational DBMSs. While there are more efficient means, none would have adequate performance given the time required to build the indexes. RSDB is able to load and rebuild the spatiotemporal index in 1:17:31 (less than simulated race time). For reasons of memory efficiency, RSDB writes its spatiotemporal index in a less than optimal manner during data load. It is then reorganized for more efficient retrieval after the load.

4.2 Playback query

To provide a measure of performance, three queries were executed. The database client sends the data to /dev/null to alleviate any influence by the output device.

The first query retrieves all data for one car in time order. For the relational DBMSs, the following query was executed:

```
SELECT * FROM car1 ORDER BY racetime;
```

The actual race time is 5064 seconds. The results, given in Table 2, show that RSDB has more than adequate performance.

4.3 Playback lap query

The second query retrieves data for a given lap for a car or a group of cars. For the relational DBMSs, a query of the following form was executed:

```
SELECT * FROM car1, car2, ...
   WHERE car1.lap=2 AND car2.lap=2 ...
      AND car1.racetime=car2.racetime ...
   ORDER BY car1.racetime;
```

The results are shown in Table 3. The actual race time for one lap is 113 seconds. PostgreSQL is marginal with 2 cars; MySQL can only handle up to 5 cars. RSDB can handle easily all 22 cars.

Table 2. Time to playback a race for one car. Time in seconds

System	Time
PostgreSQL	failed to complete
MySQL	gave error message
RSDB	406
RSDB (2 cars)	450
RSDB (3 cars)	528
RSDB (4 cars)	579
RSDB (5 cars)	612
RSDB (22 cars)	1451

Table 3. Time to playback one lap. Time in seconds

System	Time
PostgreSQL	73
PostgreSQL (2 cars)	116
MySQL	12
MySQL (2 cars)	36
MySQL (3 cars)	58
MySQL (4 cars)	81
MySQL (5 cars)	105
MySQL (6 cars)	129
RSDB	9
RSDB (2 cars)	10
RSDB (3 cars)	12
RSDB (4 cars)	13
RSDB (5 cars)	13
RSDB (6 cars)	14
RSDB (22 cars)	34

4.4 Playback lap and section

This query further qualifies the playback to a given section of the track. While times are given for all the databases, it is difficult to compare the relational databases to RSDB. For the relational DBMSs, the track was arbitrarily divided into several sections and assigned a section number. Thus the section index in the relational case is simply an index on an integer. For RSDB, the spatiotemporal index described above is used.

For the relational databases, the SQL query was of the following form:

```
SELECT * FROM car1, car2, ...
   WHERE car1.section=1 AND car1.lap=2 ...
      AND car2.section=1 AND car2.lap=2 ...
      AND car1.racetime=car2.racetime
   ORDER BY car1.racetime;
```

The results are shown in Table 4. The actual race time for the selected section is 15 seconds.

Table 4. Time to playback a specified section. Time in seconds

System	Time
PostgreSQL	46
PostgreSQL (2 cars)	52
MySQL	2
MySQL (2 cars)	6
MySQL (4 cars)	11
MySQL (6 cars)	16
RSDB	2
RSDB (2 cars)	2
RSDB (6 cars)	3
RSDB (12 cars)	4
RSDB (22 cars)	see text (14)

For RSDB with 22 cars, it took 7 seconds to locate and coordinate the data. It then took 7 seconds to transmit the data to the client. The other RSDB results have a similar character, but the times are difficult to record because they are so short.

In our implementation, data is transmitted to the client in a buffer containing all data for a single time instance. The buffer contains logical slots for each entrant. If a given car is not in the section in a given time instance, its slot contains no data.

Due to the way the spatiotemporal index works and the way we want to display data, it is necessary to build internally portions of the index from disk and coordinate the times and locations. This time is the 7 seconds stated above. Once the data is ready, it can be sent in well under the time required in the actual race. The goal is to locate the data reasonably quickly which is shown to be possible and to transmit it in less than real time (which was done).

5 Conclusion

The storage and retrieval of high speed, repetitive, unpredictable moving objects is a new area of research with new challenges. We have described a method of storing the data and an index to answer expected queries quickly. These queries are significantly different from the queries normally presented in moving object research.

The use of Formula One races in the performance study presents a difficult challenge due to the large amount of data and demands for rapid retrieval. Our implementation was able to meet the goals, whereas implementations using relational databases were not able to meet the goals.

The research suggests several areas for future study. It may be possible to link the internal nodes of the spatiotemporal index according to the object's motion. This would allow a quick look (at a coarse level) at the path taken without retrieving actual data. It may be possible to use the internal linkage to retrieve more quickly the desired nodes instead of going through the upper level X-Y matrix.

References

1. Agarwal, P., Arge, L., Erickson, J.: Indexing moving points. In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS 2000)*, Dallas, Texas, USA (2000) 175-186
2. Chon, H., Agrawal, D., Abbadi, A.: Using space-time grid for efficient management of moving objects. In *Proceedings of the Second International Workshop on Data Engineering for Wireless and Mobile Access (MobiDE 2001)*, Santa Barbara, California, USA (2001) 59-65
3. Forlizzi, L., Güting, R., Nardelli, E., Schneider, M.: A data model and data structures for moving objects databases. In *Proceedings of the ACM SIGMOD 2000 International Conference on Management of Data*, Dallas, Texas, USA (2000) 319-330
4. Güting, R., Böhlen, M., Erwig, M., Jensen, C., Lorentzos, N., Schneider, M., Vazirgiannis, M.: A foundation for representing and querying moving objects. *ACM Transactions on Database Systems* **25** (2000) 1-42
5. Kollios, G., Gunopulos, D., Tsotras, V.: On indexing mobile objects. In *Proceedings of the Eighteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS 1999)*, Philadelphia, Pennsylvania, USA (1999) 261-272
6. Moreira, J., Riberiro, C., Saglio, J.-M.: Representation and manipulation of moving points: an extended data model for location estimation. *Cartography and Geographic Information Science* **26** (1999) 109-123
7. Pfoser, D., Jensen, C., Theodoridis, Y.: Novel approaches in query processing for moving object trajectories. In *Proceedings of the 26th International Conference on Very Large Databases*, Cairo, Egypt (2000) 395-406
8. Šaltenis, S., Jensen, C., Leutenegger, S., Lopez, M.: Indexing the positions of continuously moving objects. In *Proceedings of the ACM SIGMOD 2000 International Conference on Management of Data*, Dallas, Texas, USA (2000) 331-342
9. Sistla, A., Wolfson, O., Chamberlain, S., Dao, S.: Modeling and querying moving objects. In *Proceedings of the Thirteenth International Conference on Data Engineering*, Birmingham, UK (1997) 422-432
10. Vazirgiannis, M., Wolfson, O.: A spatiotemporal model and language for moving objects on road networks. In Jensen, C., Schneider, M., Seeger, B., Tsotras, V. (eds.): *Advances in Spatial and Temporal Databases, 7th International Symposium, SSTD 2001. Lecture Notes in Computer Science.* Vol. 2121. Springer-Verlag, Berlin (2001) 20-35
11. Wolfson, O. Xu, B., Chamberlain, S., Jiang, L.: Moving objects databases: issues and solutions. In *Proceedings of the 10th International Conference on Scientific and Statistical Database Management (SSDBM)*, Capri, Italy (1998) 111-122
12. Wolfson, O., Sistla, A., Chamberlain, S., Yesha, Y.: Updating and querying databases that track mobile units. *Distributed and Parallel Databases* (1999) **7** 257-287