SharkDB: An In-memory Column-oriented Trajectory Storage

Haozhou Wang†, Kai Zheng†, Jiajie Xu‡, Bolong Zheng†, Xiaofang Zhou†, Shazia Sadiq†
†The University of Queensland, Australia ‡Soochow University, China
†{h.wang16,kevinz,b.zheng,zxf,shazia}@uq.edu.au ‡xujj@suda.edu.cn

ABSTRACT

The last decade has witnessed the prevalence of sensor and GPS technologies that produce a high volume of trajectory data representing the motion history of moving objects. However some characteristics of trajectories such as variable lengths and asynchronous sampling rates make it difficult to fit into traditional database systems that are disk-based and tuple-oriented. Motivated by the success of column store and recent development of in-memory databases, we try to explore the potential opportunities of boosting the performance of trajectory data processing by designing a novel trajectory storage within main memory. In contrast to most existing trajectory indexing methods that keep consecutive samples of the same trajectory in the same disk page, we partition the database into frames in which the positions of all moving objects at the same time instant are stored together and aligned in main memory. We found this column-wise storage to be surprisingly well suited for in-memory computing since most frames can be stored in highly compressed form, which is pivotal for increasing the memory throughput and reducing CPU-cache miss. The independence between frames also makes them natural working units when parallelizing data processing on a multi-core environment. Lastly we run a variety of common trajectory queries on both real and synthetic datasets in order to demonstrate advantages and study the limitations of our proposed storage.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications—*Spatial databases and GIS*

Keywords

Trajectory; in-memory; column-oriented data structure

1. INTRODUCTION

Driven by rapid development in sensor technology, GPS-enabled mobile devices and wireless communications, large amounts of data describing the motion history of moving objects, known as *trajectories*, are currently generated with unprecedented rate from a variety of application domains such as geographical information systems, location-based services, vehicle navigation, video tracking and so on. This calls for effective and efficient technologies to manage large scale trajectory data, which serves as a corner stone for more advanced data analytical tasks.

Even though spatial databases have been extensively studied as a research area for decades with several successful commercializations[1], they were designed to support basic spatial types only such as points, lines and polygons. Trajectories, on the other hand, are not easy to fit into a relational table with pre-defined schema since each tuple (i.e., trajectory) has different number of attributes (i.e., time-stamped points). To fix this problem, we can simply treat the entire sequence of points as a single attribute and store them in one column. Nevertheless this kind of storage will severely deteriorate query performance since it is almost impossible to utilize the spatio-temporal locality amongst trajectories. Having witnessed the limitations of existing spatial database systems, in the past decade researchers have dedicated lots of efforts in proposing novel techniques for trajectory data management. What lie in the core of these techniques are trajectory indexes, the basic design principle of which is grouping trajectory segments that are close to each other and putting them in the same node (in tree-based index) or cell (in grid index).

Owing to the development of cheap RAM-based storage technology, modern computing hardware can afford much larger main memory. It provides lots of potential opportunities to significantly improve the efficiency of big data management by shifting more or even all data from disk-based storage into main memory. This triggers numerous research interests recently on in-memory data management from both database and data mining community, ranging from memory-based indexing techniques [20] to in-memory database systems [18]. However, existing main-memory based database systems are designed for relational data, which is not suitable for storing and querying the trajectory data that possess both spatial and temporal information.

In this work, we propose an in-memory column-oriented trajectory storage, which is a read-optimized structure for storing and processing massive trajectory data. This storage can combine the merits of high throughput of main memory

[1]Many database management systems offer additional components or extensions to support spatial types and operators such as Oracle, MySQL, PostgreSQL, etc

CIKM'14, November 3–7, 2014, Shanghai, China.
Copyright 2014 ACM 978-1-4503-2598-1/14/11 ...$15.00.
http://dx.doi.org/10.1145/2661829.2661878.

and benefits of column-wise store for analytical tasks. Meanwhile, our design also supports effective trajectory data compression and query processing on the compressed trajectory data.

In order to take advantage of column-oriented data structure and compression techniques for storing and analysing trajectory data, we foresee two main challenges. The first one is how to convert trajectory data into column-oriented data structure, which not only supports varied length and multi-dimensional trajectory data, but can also perform efficient query processing in the main memory. The second one is how to deploy compression techniques on the column-oriented data structure, which can support query processing on compressed data without sacrificing much performance.

We address the above challenges by a carefully designed data structure with two-phase data processing, which combines both column-oriented data structure and compression techniques to store and analyse a huge amount of trajectory data in an in-memory database system. More specifically, trajectory allocation phase will leverage the trajectory calibration techniques [24] to calibrate the trajectory data and convert the calibrated trajectory data into column-oriented data structure. In the second phase, a trajectory encoding component will compress the trajectory data; and then encoding compressed trajectory data to make them can stored in this column-oriented data structure and support an efficient query processing on it.

Our key contributions in this work can be summarized as follows.

- We identify limitation of relational database system in handling large trajectory data.
- We develop a novel column-oriented in-memory trajectory storage, which can support compact trajectory storage in the main memory and efficient trajectory data processings.
- We deploy the system and conduct extensive experiments with several synthetic and real world large trajectory data sets. The results demonstrate that our system can achieve a high performance compares with traditional database structures, and can offer the promise of real time computing.

The rest of this paper is organized as follows. A review of related work is provided in Section 2. In section 3, we present an overview of our storage design. Section 4 and Section 5 present the details of storage structure and query processing algorithms. Section 6 reports on our experimental observations. Section 7 concludes the paper.

2. RELATED WORK

In this section, we review previous works on column-oriented data structure, in-memory data management, trajectory storage and indexing.

2.1 Column-oriented Store Architecture

Boncz et al. [5] develop a modern in-memory database system, called MonetDB, with the MIL query language to fully support the column-oriented data structure. On the other hand, the row-oriented architecture is suitable on OLTP-style applications, and Stonebraker et al. [23] indicate that system oriented towards ad-hoc querying of large amounts of data should be read-optimized. Therefore, Héman et al. [11] propose a new column-oriented data structure, called Positional Delta Tree (PDT). To improve query performance, Lemke et al. [14] propose a new algorithm to process query such as scan and aggregation operations on the compressed column-oriented data structures. Ivanova et al. [12] study a new architecture to provide an recycler intermediate in the column-oriented data structures, which is using a lightweight mechanism. Krueger et al. [13] present a linear merge algorithm to utilize the power of parallel computing for fast updating in the compressed in-memory column-oriented structures.

Unfortunately, even if these column-oriented data structure based storage systems are very powerful, they did not consider to store the spatial-temporal data such as trajectory data, which is a natural multi-dimensional data type that combine with unstructured data format. Hence, using those storage systems to store the trajectory data directly will significantly reduce the performance of query processing.

2.2 In-memory Data Management

There are several in-memory database systems have been proposed or implemented. IMS/VS Fast Path [9] is one of the earliest commercial database product. It uses the group committed for transactions to support productivity. An other earlier version of in-memory database [1, 4] based on IBM's OBE system uses points to speed up the ad-hoc transaction queries. Baulier et al. [2] implement a commercial in-memory database system, which is called DataBlitz Storage Manager. Plattner [18] indicate in-memory database can outperform the traditional database system for both OLTP (Online Transactional Processing) and OLAP (Online Analytical Processing). Subsequently, [19] implement a new column-oriented based in-memory database. Bining et al [3]. propose a dictionary-based order-preserving string compression algorithm in the in-memory column system. On the other hand, Rao and Ross [20] propose CSB+-trees as the main-memory index structure, which is a CPU cache optimized method and is improved from B+-trees. At the same time, a cost model for in-memory database system is proposed by Manegold et al. [15] , which can provide an accurate cost functions for database operations.

Due to the underlying use of the relational model in these in-memory database systems, analytical trajectory queries cannot be well-supported.

2.3 Trajectory Storage and Indexing

The most popular and classical data structure for spatial data is R-tree [10]. However, R-tree is not good enough to support trajectory data, since it is designed for spatial information only. Therefore, many optimizations are proposed to make the R-tree based structures support the trajectory data. TB-tree [17] uses a hybrid tree structure to store and index both spatial and temporal information, but is not adequate to process long trajectories, which can make the bounding rectangles very large. TPR-tree [22] and TPR*-tree [25] invoke the predication model to predict the future positions of moving objects. On the other hand, partitioning trajectories into segments become a new way to improve the query performance. Rasetic et al. [21] derive an analytical cost model to control the splitting process for a trajectory into segments based on given query. SETI [7]

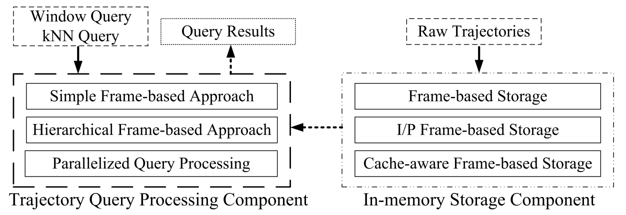

Window Query
kNN Query
Query Results
Raw Trajectories
Simple Frame-based Approach
Hierarchical Frame-based Approach
Parallelized Query Processing
Frame-based Storage
I/P Frame-based Storage
Cache-aware Frame-based Storage
Trajectory Query Processing Component
In-memory Storage Component

Figure 1: Storage architecture

stores trajectory segments in a 3D R-tree for their spatial information. Meanwhile, SETI indexes the temporal information by using one dimensional time lines to increase the search performance. PIST [6] partitions the sample points rather than partitioning the trajectories. Similarly, the TrajStore [8] propose a new adaptive storage system that indexes the trajectory data based on quad-tree index and clustering methods.

These algorithms or systems are designed based for disk based systems, which means I/O cost between hard disk to memory is the main concern. However, there is no I/O cost in in-memory based systems. Thus than algorithms and systems can not easily aligned for in-memory systems.

3. STORAGE ARCHITECTURE

The storage architecture is illustrated in Fig 1, which includes two main parts: the real-time query processing component and the in-memory data storage. The in-memory data storage system encodes the raw trajectories and stores them in a re-designed column-oriented data structure. The real-time query processing component can support two fundamental queries including window query and nearest neighbour query in real time. We give a brief overview of these two parts in the following:

In-memory Storage: In the in-memory data storage component, we create a novel frame based column-oriented data structure to store the trajectory data. Hence, we propose three storage models, frame-based storage, I/P frame-based storage and cache-aware frame based storage, to convert and calibrate the raw trajectory data into this new frame based data structure. The last two models also embed trajectory compression algorithm to reduce the data footprint in the memory. We will discuss this part in Section 4.

Real-Time Query Processing System: We implement three query processing approaches to support real-time query processing in the system. The simple frame based approach is working over frame data structure; and hierarchical frame based approach builds a multi-level frame data structure from in-memory storage component to achieve better performance. Moreover, we extend previous approaches to allow parallel query processing. We will discuss this part in Section 5.

4. IN-MEMORY STORAGE

In this section, we detail the three data structures as well as its encoding algorithm to store the raw trajectory data into the in-memory column-oriented data structure.

4.1 Frame-based Storage

Different from basic operation (i.e., selection or deletion), most analysis tasks such as window query and nearest neighbour query only need to touch few trajectories to get answer. However, the row-oriented data structure is not good for this task, since the whole table scanning is hard to avoid during trajectory query processing, if these data are not indexed. On the other hand, the column-oriented data structure is known to have better performance in analysis task that compares with row-oriented data structure [18]. To get this advantage, we propose a novel frame based column-oriented data structure to store trajectory data in the main memory. Thus, we create a sequence of frames from trajectory data directly. In each frame, we assign a particular time interval, then allocate each trajectory sample point to the related frame based on its recorded time. Therefore, a frame contains all of trajectory sample points in the whole dataset, which are recorded at the time interval of the frame. And such time interval is named as frame rate. For instance, as Fig. 2 shows, if we set the time interval is 1 minute (i.e. the frame rate is 60 seconds), the time period from 9:01 to 9:05 is split into 4 frames. Hence, the sample points in the trajectory $T_1 = p'_1, p'_2, p'_3, p'_4$ and $T_2 = p_1, p_2, p_3, p_4$ will be aligned to the related frame, in this example, p'_1, p_1 and p'_2, p_2 are assigned into $Frame1$ and $Frame2$ respectively.

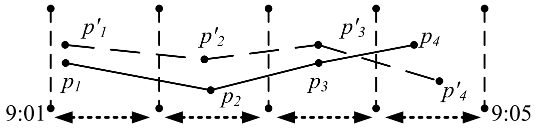


Figure 2: Trajectory snapshot

To keep each frame simple and tidy, each frame contains no more than one point for each trajectory. To make this frame structure continuous in temporal space, each frame must be strictly synchronized by same time period (i.e., with same frame rate). Normally, the frame rate will be same as sampling rate of trajectory. However, a trajectory in a raw trajectory dataset may have different sampling rate compared with other trajectories due to unstable GPS signal. Therefore, this situation can make the whole trajectory dataset become heterogeneous, which may affect the frame structure. To avoid this issue, if there is more than one sample point, which belongs to the same trajectory, they are aligned to the same frame. We calculate the SED [16] distance of each sample point; and keep the sample point with largest SED distance and remove the rest sample points in that frame. This is because such a point contains more information than other points that have smaller SED distance. In the same time, if there is no sample point of a trajectory in a frame, which between its start frame and end frame, we use line interpolation method to add a new sample point of the trajectory in that frame. Moreover, we conduct a set of accuracy experiments in section 6 to show the influence of heterogeneous trajectories.

Table 1: Example of Frame based Structure

9:01-9:02	9:02-9:03	9:03-9:04	9:04-9:05
Frame 1	Frame 2	Frame 3	Frame 4
(1,p'_1)	(1,p'_2)	(1,p'_3)	(1,p'_4)
(2,p_1)	(2,p_2)	(2,p_3)	(2p_4)

After allocating, converting each frame into column-oriented data structure become straightforward. Basically, we only need to insert each frame as a single column into the in-memory database. The ID of each column is named as its own time interval (e.g., 9 : 01-9 : 02) as Table. 1 shows. In addition, each sample point (i.e., the object) in column contains its trajectory ID, longitude and latitude. After aligning, we do not need the timestamp information of a sample point any more, since the temporal information can be recovered by the column ID. The advantage is that our frame data structure is a natural column-oriented data structure and synchronized by time, hence, storing this data in the column-oriented structure is simple.

4.2 I/P Frame-based Storage

For in-memory database, it is good to utilize compression techniques to reduce the size of data and improve query performance, since the compressed data can reduce the footprint of data in the memory, which can in turn reduce the searching time in the memory. For trajectory compression, we consider delta encoding technique to compress the trajectory data, which keeps information for the first point and using the δ value to record the information for the rest of points. Therefore, the delta encoding will not change the sampling rate of trajectories and it allows the storage system to use less bits to store the trajectory data. However, a problems is raised when using delta encoding. For each column, the type of object (point) must be the same, which means the objects in a single column can only be original point or only be delta encoded point. However, the start time of trajectory and the length of trajectory is arbitrary, so we can not simply use delta encoding. Therefore, we propose a novel I/P frame based data structure to reduce the memory consumption without this problem.

Building an I/P frame data structure includes grouping process and encoding process. First of all, we split the whole sequence of frames into small groups, called frame group, and each frame group contains n continuing frames. Based on the example in Fig. 1 , if we set $n = 4$, and the $Frame1$ to $Frame4$ are allocated in a group. After encoding phase, for each group, we keep the first frame raw information as an I-frame, and the subsequent frames, called P-frame, use delta encoding to be compressed. The example is shown in Table. 2, the $Frame1$ is kept as I-frame IF_1. The rest of the frames $Frame2$ to $Frame4$ as P_1 to P_3, then we encode the sample points in P_1 by calculating the difference between IF_1 and its related sample point in P_1 as a P-frame point $P_1.PF_1$. Again, we continue encoding the sample points in P_2 and P_3 as P-frame points $P_2.PF_1$ and $P_3.PF_1$ respectively. At the end, we get one I-frame column and 3 encoded P-frames columns as shown in Table. 2.

Table 2: Example of I/P-frame Structure

Frame 1	Frame 2	Frame 3	Frame 4
(1,p)	(1,Δp)	(1,Δp)	(1,Δp)
(2,p)	(2,Δp)	(3,Δp)	(4,Δp)
(3,p)	(3,Δp)	(4,Δp)	
(4,p)	(4,Δp)		

However, this I/P frame encoding algorithm still has some drawbacks. For reconstruction of a segment of trajectory that is contained in a frame group, current solution has to scan all of columns in that group, which includes both I-frame and P-frame; and this process is very expensive, since it has to scan a lot of memory. Therefore, the performance of query processing can be reduced; and it does not fully utilize the power of column-oriented data structure.

4.3 Cache-aware Frame-based Storage

To increase the speed of trajectory reconstruction performance, the number of sequential scans for columns should be minimized during query processing. The idea of avoiding the sequential scan in all of columns in a frame group is to reduce the number of scans in P-frame, since the P-frames are highly related to I-frames. Therefore we fix the length of both I-frame column and P-frame columns in a group, which means the length of each P-frame column equals to the length of I-frame column. As a result, for the same segment of trajectory in a frame group, each P-frame point of that trajectory shares the same index with the point in I-frame. Meanwhile, if a trajectory segment has not full filled a frame group such as a trajectory segment has 1 I-frame point and 2 P-frame points in a $n = 4$ frame group, we use a Nil code as a place-holder in the rest of P-frames. Hence, to reconstruct a trajectory segment, we can scan the I-frame point in I-frame only and access the P-frame points in P-frames directly in the memory by adding the offset from I-frame point. Moreover, in each P-frame point, the trajectory ID is no longer needed, since it can be recovered from I-frame point. For example, as Table. 3 shows, to reconstruct trajectory T_4, we scan I-frame first to get its I-frame point, then we can access the related P-frame points directly by adding an offset from the beginning address of each P-frame column. This can improve the speed significantly during reconstruction processing.

Moreover, in modern architectures of computer system, accessing data from memory to CPU is via a hierarchical memory structure. Typically, CPU has built-in cache memories, which is connecting to system main memory directly. On the other hand, the reading action from memory to CPU is parametrized by CPU cache line size (typical is 64 bytes), which is the basic transferring unit. For a reading action, CPU will read 64 bytes data from memory no matter these data could be fully used or not. Therefore, to increase the performance, it requires to fill up the CPU cache line as much as possible with useful data for each memory access to reduce the total number of accessing from CPU. However, this task is challenging as we need to store the P-frame point of a trajectory segment continuously and cannot break the column-oriented data structure at the same time.

Table 3: Example of Cache-aware Frame-based Structure

9:01-9:02	9:02-9:03	9:03-9:04	9:04-9:05
(1,p)	(1,Δp)	(1,Δp)	(1,Δp)
(2,p)	(2,Δp)	Nil	Nil
(3,p)	(3,Δp)	(3,Δp)	Nil
(4,p)	(4,Δp)	(4,Δp)	(4,Δp)

Hence, we use a hybrid data structure to store the P-frames in a frame group as they are highly related. Our hybrid data structure is based on a two dimensional array, which is created as an $array[x][y]$, where x equals to the number of I-frame points in this frame group, and y equals to the number of P-frame columns in this frame group. So, the value of x in array is the index of I-frame points; the value of y indicates the column number of P-frame group; and the object at $array[x][y]$ is the P-frame point. For example, $array[1][2]$ is the P-frame point $PF_{1,2}$, which is shown in Fig. 3, and 1 for first I-frame point and 2 for second P-frame column. As a result, CPU can fetch an I-frame point with its related P-frame points and fill in a cache line block by a single memory accessing. When CPU tries to access the next P-frame point, it is already in the CPU cache, and CPU can access them directly without needs to search memory again. Consequently, it will increase the performance of query processing. For example, as Fig 3 shows, assuming a memory access reads 2 points vertically from P-frame array. It will cost two memory seeking requests to decode a trajectory segment in a frame group for non-cache optimization data structure, since the other points in the cache line is not related to this segment and is not usable. On the other hand, it only needs to cost one memory access for decode this trajectory segment in the cache optimized data structure, since the related four points can be read from CPU cache directly.

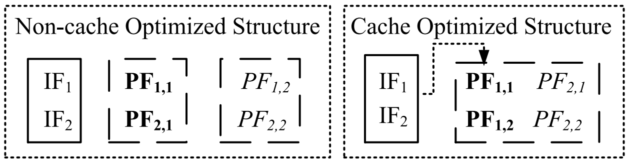


Figure 3: Example of Cache-aware I/P-frame Structure

5. TRAJECTORY QUERY PROCESSING

There are two fundamental query types, window query and top-k nearest neighbour (kNN) query. For window query, it will find all trajectories in the data set that are active during a given period and passing a given region. For kNN query, it will find top-k trajectories in the trajectory data set that are close to a given point and active during a given period of time. In this section, we propose simple frame based approach and hierarchical I/P frame data structure based approach proposed to answer these queries. A parallel version for such approaches is discussed in section 5.3.

5.1 Simple Frame based Approach

For query processing, it is necessary to reduce the number of reconstruction trajectory segments in the frame group as much as possible. In other words, the unnecessary frame groups should be pruned as much as possible. Therefore, we deploy a two phase processing that includes pruning and refining, which are shown in Algorithm. 1. In the pruning phase, we first filter out the frame group columns, which are out of the given time period (Line 1). For I-frame point (i.e., original trajectory sample point) in each frame group, we calculate the maximum moving distance D_{max} for its P-frame points members (Line 2-4). The D_{max} equals to user defined maximum possible moving speed multiple the time duration of this frame group. Then we use D_{max} as the boundary value of this trajectory segment for pruning (Line 5) based on different query types (i.e., Window query or NN query). In the refining phase (Line 6-8) , we decode and reconstruct trajectory segments from I-frame point with its related P-frame points and find the final answer based on given query type. More specifically, we detail the difference of query processing between window query and kNN query.

Window Query: For window query, at the pruning phase (Line 5), we use D_{max} to estimate the moving area of its related frame group (i.e., trajectory segment). If the moving area is overlapping with query window, we decode the P-frame points to find the actual answer of window query (Line 7-8).

Algorithm 1: Frame Search Algorithm

```
   Input: Query $Q$, Frame structure $F$
   Output: Results set
1  $ColumnIds \leftarrow$ TimeInterval($Q$, $F$);
2  for $i \leftarrow ColumnIds.start$ to $ColumnIds.end$ do
3      foreach frame group $FG$ in $FrameColumn.get(i)$ do
4          $D_{max} \leftarrow$ CalculateDistance($FG$, $maxSpeed$);
5          if Check($D_{max}$, $Q$) then
6              $Tr$ = Decode($FG$);
7              if CheckTrajectory($Tr$, $Q$) then
8                  Update($Tr$);
9              end
10         end
11     end
12 end
```

kNN Query: For the kNN query, we calculate the distance from I-frame point to query point and minus the D_{max} as the lower bound. If the lower bound is larger than the current k^{th} best true distance, which means this trajectory segment cannot be closer than current best examined trajectory, we can prune it safely (Line 5). Otherwise, we put this trajectory segment into refining phase (Line 6-8) to find its true distance to the query point. Moreover, since we are processing on each frame group (i.e., the trajectory segment), not for the complete trajectory, we need to avoid the different frame groups with same trajectory ID to be pushed into result set. This situation can lead several trajectory segments with same trajectory ID (i.e., these segments belongs to one trajectory) to be consider as final results. It can cause the number of final results (i.e., the returned trajectories) are less than k, which make this query fail. Therefore, we extend the priority heap to a hashed priority heap, which its elements are hashed by trajectory ID. When updating a new frame group, the system will first check the elements in the results set whether there is an element containing same trajectory ID as the new frame group. If it exists, system will only update the existing element by using this new frame group rather than creating a new element in the results set.

5.2 Hierarchical Frame based Approach

During the query processing, the most time consuming part is the sequential searching in the I-frame columns. To reduce the searching time, we propose a new hierarchical frame structure that expand from simple frame based approach.

As Fig. 4 shows, we build this structure from bottom to top, which is based on the sample I/P frame data structure to reduce the number of I-frame columns visited during query processing. To reduce the complexity of the whole multi-layer structure, we still use frame group as the basic unit in the structure. First of all, we extract the I-frame columns from current top level to build a new I/P frame structure layer, which becomes an upper level of current level. Then we assign every n I-frame columns into a frame group, where n is the number of frames in a frame group. After this, we use the same encoding methods that are proposed in the Section 3 to encode the I-frame columns in the new frame groups. That means, in each new frame group, we keep the first I-frame column unchanged as the new I-frame column and re-encode the rest I-frame columns as the new P-frame columns. In Fig. 4, an example is shown the process to build level 2 structure from level 1 structure and build level 3 structure from level 2 structure respectively. During the processing, we keep building new layers until the time duration of frame group is larger than the average time duration of trajectories at current level. This is because if we do not stop building at this level, most trajectories will be represented as a single frame point at the upper level, and such level cannot assist in improving query processing.

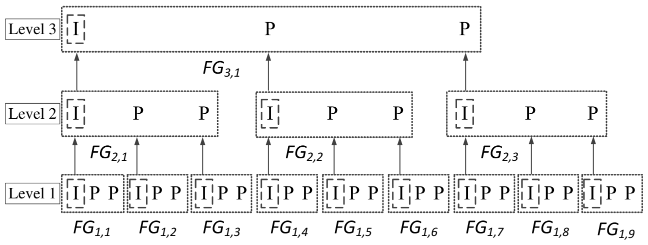


Figure 4: Hierarchical I/P-frame Data Structure

Although the encoding technology can reduce the space consumption substantially, keeping the I-frame columns information in each level is still consuming space and will limit the number of layers in hierarchical structure due to the memory space limitation. When we build a new layer, for each new I-frame point that needs to copy from lower level, we only copy the memory address from original I-frame point instead of copy full information. The full information of such I-frame points are kept in the bottom level only. Therefore, we can reduce the memory consumption for this data structure. In addition, we use the same strategy to copy/encode new level P-frame points. Hence, when algorithm accesses such P-frame point, it can get the information directly without need to decode this P-frame point; and can increase traversing speed of algorithm on this data structure.

At the same time, in the high level of hierarchical structure, the frame points of each trajectory become sparse, which leads to not only making the time duration between each frame column very large, but also the distance between two frame point becomes longer. Hence, it is easy to see the bound D_{max} will be too large to do the pruning. This is because a large value of D_{max} will increase the searching space dramatically, where can reduce the performance of query processing or even make it same as the exhausted search. To solve this issue, for each frame group started at the bottom level, we calculate its MBR information and embed it into its I-frame point to instead of calculate the D_{max} during query processing. The upper level MBR information is calculate based on the MBRs information of its lower level frame groups to keep the upper level MBR covers all of MBR in the lower level frame groups used to build this frame group. For example, the MBR of $FG_{2,1}$ is the superset of MBRs that include frame groups from $FG_{1,1}$ to $FG_{1,3}$.

In addition, since the structure of each layer in whole hierarchical structure is the same, we can keep our two phase processing with fewer changes. For traversing the hierarchical structure, we use the best first algorithm as shown in Algorithm.2. Firstly, we initiate a priority heap and push the first level frame groups within query time range into it (Line 1). Then if a popped frame group is already in the bottom level, we use the algorithm same as Algorithm. 1 to update the results set (Line 3-4). Otherwise, we travel into next level by decoding the popped frame group directly, and filter out decoded frame groups in next level are out of time range of query (Line 6-7). Finally, we exam filtered frame groups and update the priority queue in Line 8-9. Now we detail the process for each type of query:

Algorithm 2: Hierarchical Frame Search Algorithm

```
   Input: Query $Q$, Hierarchical Frame structure $HF$
   Output: Results set
1  $PQ \leftarrow$ initial($HF$, $Q.timeRange$) ;
2  while $FG \leftarrow PQ.pop()$ do
3      if $FG$ at bottom level then
4          Same as Algorithm. 1;
5      else
6          $framegroups \leftarrow$ Decode($FG$, $HF$);
7          foreach frame group FG in $framegroups$ do
8              if Check(FG, Q) then
9                  $PQ.push(FG)$;
10             end
11         end
12     end
13 end
```

Window Query: For each selected frame group, we check the MBR information in its I-frame point. If the MBR is overlapping with query window, we decode them. For example, in Fig. 4, if frame group $FG_{3,1}$ at level 3 is popped from candidate list, we then select frame group from $FG_{2,1}$ to $FG_{2,3}$ at level 2, which are used to build frame group $FG_{3,1}$. The refining phase is same as simple frame based approach (Line 4).

kNN Query: For the kNN query, we use the minimum distance from MBR to query point as the new low bound for pruning phase. For $Check()$ function, we calculate its low bound and push it into the priority heap. Same as Algorithm. 1, if a frame group is already in the bottom level, we calculate this true distance from query point to this frame group. If the true distance is less than best so far distance, we push this frame group with its true distance into our hashed priority heap as update in the results set.

5.3 Parallel Computing

A general configuration of modern servers typically contains multiple CPUs with multi-cores built-in in each CPU. Therefore, it now becomes the standard that supports parallel computing for query processing, which can fully release the power of parallel computing. The problem of paralleling query processing is like a divide and conquer problem, for instance, how to divide a query into sub-queries to send to different threads and how to combine the sub-results into final result. Based on the advantage of I/P frame data structure, where each frame group is highly independent and "share nothing" with each others, the divide and conquer process is simple, and very stable. Algorithm. 3 shows the parallel query processing on hierarchical structure. Firstly, we create a thread pool to manage threads (Line 1). Then we assign each frame group column into different threads to process and push results to the queue (Line 2). After this, we do parallel decoding to get the next level frame groups from popped frame group and assign them into different thread. In each single thread, we run Algorithm. 2 to check and keep traversing strategy. After this, we merge the results from each single thread. Now we detail our approach for parallel window query, and parallel kNN query.

Algorithm 3: Parallel Search Algorithm

```
Input: Query $Q$, Hierarchical Frame structure $HF$
Output: Results set
1 $TP \leftarrow$ thread initial ;
2 $PQ \leftarrow$ parallelInitial($HF$, $Q.timeRange$, $TP$) ;
3 while $FG \leftarrow PQ.pop()$ do
4     parallelDecoding($FG$, $HF$, $TP$);
5     Run Algorithm. 2 in each single thread;
6     Merge results from each thread and do Update();
7 end
```

Window Query: To store the results in parallel, we create an $array[x][y]$ to organize the result trajectories; and each $array[x]$ represents a result trajectory and each $array[x][y]$ represents a sample point of a result trajectory. If a thread finds its results containing new trajectories (new IDs), it will create new arrays in this array. When creating the new array for a new result, this thread locks this object to block new array creation request from other threads until finished, which is to avoid creating multiple arrays for one trajectory.

kNN Query: The challenge of parallelization kNN query is that the value of the best so far (the minimum distance of from current candidate trajectory to query point) needs to keep updating efficiently to shrink the spatial search area during query processing. Therefore, during query processing, each single thread uses the current best so far value for pruning and returns a new trajectory id with its distance if any better candidate is found. Once a single thread finds a better candidate, it first applies a lock on both the best so far variable and the hashed priority queue. Then, this thread update the best so far variable if the distance of this candidate is better than current value; and updates the hashed priority queue. At the end, this thread releases the locks and sends a signal to system to wait for assigning next frame group. System continues assigning frame group(s) to free threads in the thread pool until all the frame groups in the candidate list are investigated.

6. EXPERIMENTS

In this section, we conduct extensive experiments on real trajectory datasets and a synthetic trajectory dataset to study the performance of the proposed data structure and query processing methods.

6.1 Experimental Setup

We use two real world trajectory datasets, which are collected from two big cities. Similarly, for testing the influence of heterogeneous trajectory dataset in our system, we generate a synthetic trajectory dataset; and the sampling rate distribution will be based on normal distribution with different mean and standard deviation of sampling rate The results are discussed in Section 6.3. The detailed statistics of the datasets are given in Table 4.

Table 4: Trajectory Dataset

Dataset	# Sample Points	# Trajectories	Size
Dataset A	0.7 billion	243,194	14.6 GB
Dataset B	80 million	28,784	2.52 GB
Synthetic	3 billion	729,582	80GB

We compare the time cost of the proposed column-oriented data structures against the row-oriented data structure. The row-oriented data structure is segmenting the trajectory and storing them in-memory, which is implemented by ourselves and called segmented trajectory database. In the segmented trajectory database, a trajectory is split into small segments, and each segment contains a fixed number (or no more than a fixed number) of sample points. Each segment is described as a MBR, which also includes its start time and end time to speed up the query processing. For number of sample points in each segment, we experimented with different values, and finally chose 50, which achieves the best performance in our experiments.

Table. 5 shows the default value and the range of each parameters. Due to the space limitation, we only examine the kNN query with $k = 5$, since we are more interested in how length of time interval of query can affect the performance. The T_s is the sampling rate of the trajectory datasets, so $2T_s$ means the time interval of each frame is two times with trajectory sampling rate. The default value of time interval for frame is T_s, which is to avoid the accuracy issue to make a fair competition with baseline method. For our hierarchical based approach, we set the building stop condition as equal to the time duration of each frame group when it reaches average length of trajectory. Meanwhile, we use 10 cores for parallel model testing. For each set of experiment, we generate 100 queries and calculate the average running time. Each query is generate randomly with selected value of parameters. All the algorithms including segmented trajectory database are implemented in Java and run on a sever with two Intel 8-cores CPUs and 192GB memory.

Table 5: Parameters Setting

Parameter	Default Value	Range
# frames per frame group n	16	8-32
time interval per frame	T_s	$1/4T_s$-$3T_s$
area of spatial window (AS)	3%	1%-11%
length of time interval (TI)	3h	1h - 11h

6.2 Performance Evaluation

6.2.1 Frame Storage Evaluation

In this section, we compare the different performance between segmented trajectory database, frame storage, I/P frame storage and cache-aware I/P frame storage on real world datasets. To minimize the effect from query processing approach, for I/P frame storage and cache-aware I/P frame storage, we only use the simple frame based approach; for frame storage, we apply sequential searching on each frame column.

Performance of Window Query We first evaluate the efficiency of these four approaches with different window radius with default time interval. The results are shown in Fig 5, we can see that the performance of column-oriented based data structure are very stable. This is because the main cost of query processing in column-oriented based data structure is searching on the I-frame columns; and the number of I-frame columns that need to be searched are fixed since the time interval of queries are the same in this experiment. Hence, the performance of I/P frame storage is better than frame storage, since less I-frame columns need to be searched. Moreover, the cache-aware I/P frame storage optimizes the decoding performance, so it gets the best performance. Then, we conduct another experiment by changing the size of time window of each query, the results are also shown in Fig 5. Based on results, we can see that the length of time interval is the main factor that can affect query performance, especially for column-oriented based data structure, since the length of time intervals decide how many frames need to be searched in I/P frame based storage.

Performance of kNN Query We investigate the query performance with regard to the length of time interval for each query. The results are shown in Fig 6. Similar to window query, the length of time interval of each query is also the main factor to affect query performance. Moreover, different to window query, the kNN query needs to investigate more frame points to get the final answers. Therefore, for I/P frame based storage, the cost of decoding P-frame points can also be much larger than window query. At the same time, the cache-aware I/P frame storage can reduce the decoding cost as it is optimized for CPU cache to increase decoding performance. Hence, the cache-aware I/P frame storage is much faster than I/P frame storage, when compared with window query. Finally, we can see that the I/P frame storage can improve the performance to at least 10 times faster than row-oriented database.

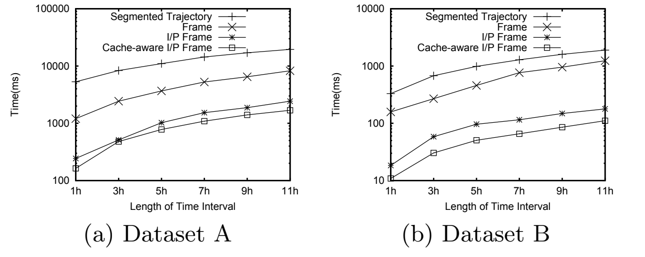


Figure 6: Performance of kNN Query

Effect of Frame Rate Next we study the query performance when the frame rate is varying. We use the running time of kNN query on the dataset A as the measurement. As the frame rate will cause the same effects in the three frame storages, we only examine the cache aware I/P frame storage. The results are present in Fig 7. Without considering the accuracy, the experiment with parameter setting $3T_s$ has the best performance since the size of whole dataset has been reduced significantly. In addition, the increasing frame rate will make the distance between two sample point longer and increasing search area of each frame group. Therefore, it causes more candidate frame groups to be pushed into candidate list and reduce the performance in the refining phase.

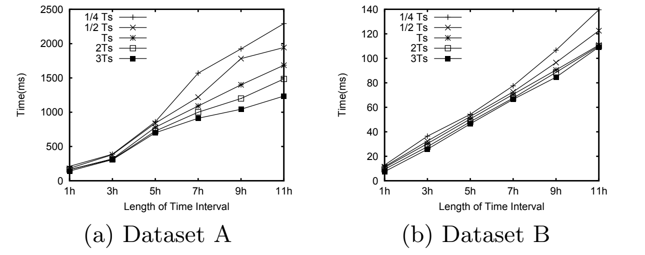


Figure 7: Effect of Frame Rate

Effect of Number of Frames for each Frame Group Finally, we evaluate the query performance with different n. We use the same measurement method as the effect of frame rate. The running time is reported in Fig 8. As frame storage do not have frame group structure; and both I/P frame storage and cache-aware I/P frame storage use the same frame group structure, we also only examine the cache-aware I/P frame storage. We can find the performance is increasing when the number of n is increasing. This is because, in the query processing on P-frames, the system can access the P-frame points directly without need to search the P-frame columns and send to CPU to decode, which can reduce the search time and increase the performance. Meanwhile, the system can get more benefits for the compression technical, which can reduce the consumption of memory bandwidth.

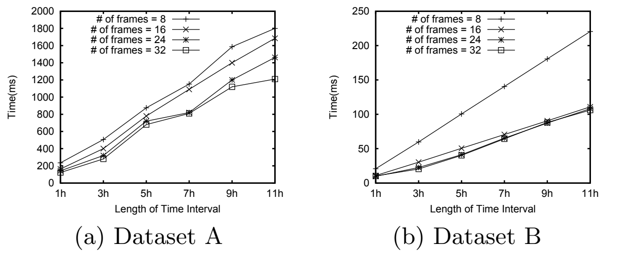


Figure 8: Effect of Number of Frames

6.2.2 *Query Processing Evaluation*

In the next set of experiments, we look at the different performance of our query processing approaches, which are simple frame based approach, hierarchical frame based approach and hierarchical frame based approach with parallel computing. We will test the performance of window query and kNN query.

Performance of Window Query Similar to previous experiments, the results are illustrated in Fig 9. Without surprise, the hierarchical frame based approach is better than simple frame based approach, which shows using the hierarchical structure and MBR can reduce the search space significantly. On the other hand, the parallel computing technology can improve the performance when the time interval is increasing. This is because, the overhead cost of parallel computing is more than single thread, since it has to create a thread pool to manage the threads at beginning, which leads to lower increase if the investigated I-frame columns are small (i.e., in the short length of time interval). But this overhead cost is constant, so the parallel computing approach can achieve large increasing of performance when the search space is large.

Performance of kNN Query We now evaluate the performance of kNN query with changing the length of time interval, which is shown in Fig 10. We can see the hierarchical frame based approach can improve the performance around 30%. As the kNN query needs to investigate more frames, the parallel computing approach boosts the performance close to the theoretical performance improvement, which is near 10 time faster compared with single thread.

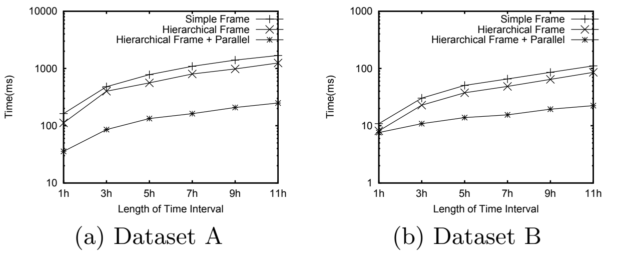
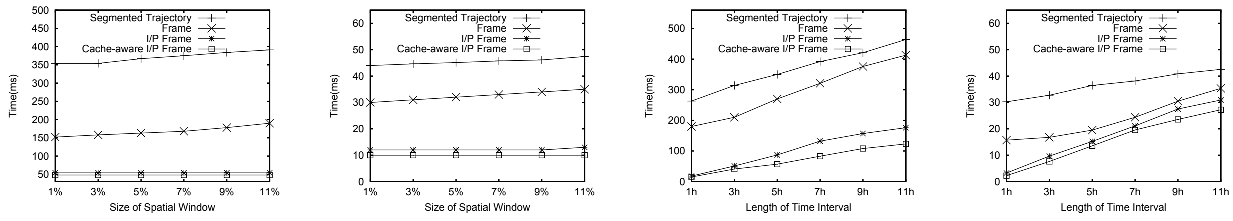


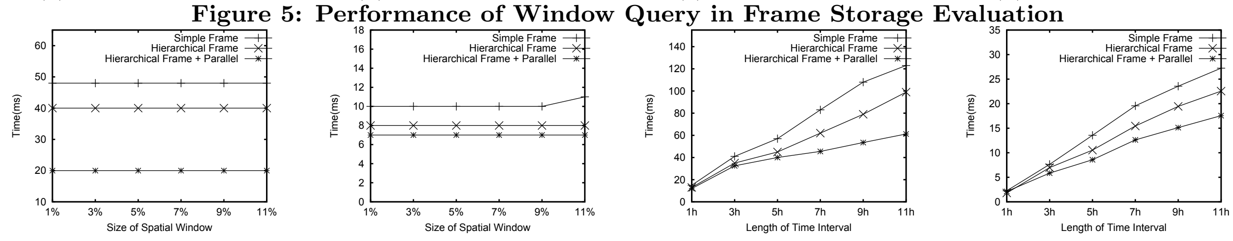
Figure 10: Performance of kNN Query

Scalability Evaluation We also evaluate the scalability of all the approaches under two queries/operations. To



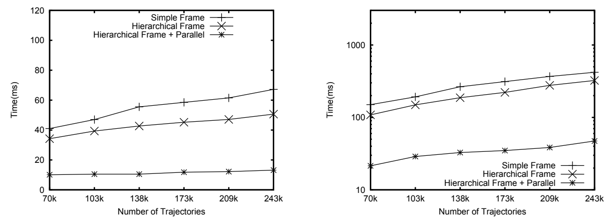
(a) AS in Dataset A (b) AS in Dataset B (c) TI in Dataset A (d) TI in Dataset B

Figure 5: Performance of Window Query in Frame Storage Evaluation



(a) AS in Dataset A (b) AS in Dataset B (c) TI in Dataset A (d) TI in Dataset B

Figure 9: Performance of Window Query in Query Processing Evaluation



(a) Window Query (b) NN Query

Figure 11: Scalability

achieve this, we random select trajectories in the dataset A with different number from 70k to (approx.) 243k. Therefore, when we increase the number of trajectories, the density of trajectory in certain area (i.e., the area is covered by whole trajectory dataset) also be increased to the higher value. The running time is reported in Fig 11. It is illustrated that the time cost of all three storages increase linearly/sub-linearly with respect to the size of dataset. Hierarchical frame based approach with parallel computing achieves the best performance in this experiment.

6.2.3 Accuracy and Compression Ratio Evaluation

The accuracy testing is designed for frame storages particularly, as we mentioned above, time interval of each frame (i.e., frame rate) will affect the accuracy in frame encoding methods if time interval of each frame is larger than sampling frequency. At the same time, the compression ratio is calculated by the *size of encoded data in the memory* divided by the *size of original data in the memory*

Effect of Frame Rate We evaluate the accuracy by changing time interval of each frame. To measure the accuracy, we first randomly select 200 points; and find their kNN trajectories as ground truth, where $k = 100$. Then we use these points as the query points to do same query processing on the our approach. After we get the results, we use recall to measure the accuracy. For example, for a query point, its kNN ($k = 5$) ground truth is $1, 2, 3, 4, 5$, and result of our approaches is $1, 2, 3, 4, 6$, and its recall is 80%. The results are shown in the Table 6 with the compression ratio. As expected, increasing time interval of frame will cause a low accuracy rate, but the compression performance is increased in this case. Moreover, when the time interval of each frame is less than sampling rate of trajectory, there is no improvement in accuracy, but the need for memory is increased.

Table 6: The results of Accuracy

	Dataset A		Dataset B	
Time interval	Recall	Ratio	Recall	Ratio
$1/4\ T_s$	100%	69.3%	100%	69.3%
$1/2\ T_s$	100%	36.3%	100%	36.3%
T_s	100%	18.3%	100%	18.3%
$2T_2$	82%	11.2%	86%	11.2%
$3T_3$	76%	7.4%	77%	7.4%

Effect of Number of Frames for each Frame Group Then we investigate the compression ratio with regard to the number of frames in a frame group. The results are shown in Table. 7 for both datasets. We observe that for different value of n, definitely, the higher n is set, the higher compression ratio we can achieve. This is because increasing n will reduce the total number of I-frame columns in the whole storage system; and P-frame point costs much less than I-frame point.

Table 7: The Results of Compression Ratio

	$n = 8$	$n = 16$	$n = 24$	$n = 32$
Dataset A	21.1%	18.3%	16.1%	15.0%
Dataset B	21.1%	18.3%	16.1%	15.0%

6.3 Synthetic Trajectory Data Evaluation

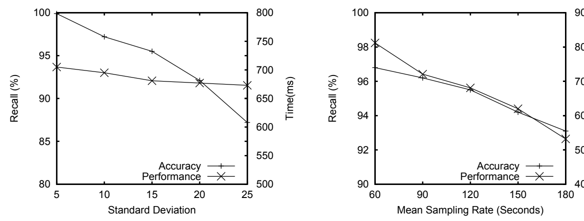
The range of mean and standard deviation of sampling rates is shown in Table. 8. Meanwhile the range of sampling rate is from 30 seconds to 240 seconds during generation. Moreover, in this experiment, we set the frame rate equal to the mean sampling rate, and use the same settings (kNN query) as previous experiment for accuracy and performance testing.

Table 8: Synthetic Evaluation Parameters Setting

Parameter	Default Value	Range
mean sampling rate (seconds)	120	60-180
standard deviation	15	5-25

Effect of Standard Deviation Figure. 12(a) illustrates the effect of standard deviation of sampling rate. As we can see, the heterogeneous trajectory can cause an impact on accuracy of query processing, especially when the standard derivation is high. This is because, if sampling rate of a part of the trajectory is higher than mean sampling rate, some sample points will be removed during frame encoding. Therefore, the higher standard deviation, the more sample points will be removed. This case will decrease the accuracy rate, which is similar as the evaluation of effect of frame rate. However, we can see that our system has a good tolerance for low standard deviation. On the other hand, there is no effect on performance for different standard deviation since it does not affect the frame rate.

Effect of Mean Sampling Rates Figure. 12(b) shows the results of different mean sampling rates. We can find that performance is increasing with lower mean sampling rate, since less frame group columns will be scanned during query processing. This is because, the larger mean sampling rate has longer frame rate, which reduces the number of frame group column in a certain time period. In the same time, based on our evaluation settings, the smaller mean sampling rate can increase the accuracy. The reason is the smaller frame time interval in each frame, which means the number of sample points that need to be removed is lower than the larger frame time interval for high sampling trajectories.



(a) Standard Deviation (b) Mean

Figure 12: Synthetic Evaluation

7. CONCLUSION

In this paper, we presented SharkDB, a new in-memory column-oriented storage system for storing and querying trajectory data. We developed an I/P frame based column-oriented data structure with CPU-cache optimization to provide an efficient storage system. To support efficient query processing, we proposed several algorithms that using both hierarchical structure and parallel computing, which fully utilize the of I/P frame data structure. Extensive experimental results based on both real and synthetic dataset demonstrate that the proposed method outperforms several baseline algorithms significantly with good scalability.

8. ACKNOWLEDGEMENTS

This research is partially supported by the Australian Research Council (Grants No.DP110103423, DP120102829 and DE140100215) and Natural Science Foundation of China (Grant No.61232006).

9. REFERENCES

[1] A. C. Ammann, M. Hanrahan, and R. Krishnamurthy. Design of a memory resident DBMS. In *COMPCON*, pages 54–58, 1985.

[2] J. Baulier, P. Bohannon, S. Gogate, C. Gupta, and S. Haldar. DataBlitz storage manager: main-memory database performance for critical applications. In *SIGMOD*, pages 519–520, 1999.

[3] C. Binnig, S. Hildenbrand, and F. Färber. Dictionary-based order-preserving string compression for main memory column stores. In *SIGMOD*, pages 283–296, 2009.

[4] D. Bitton, M. Hanrahan, and C. Turbyfill. Performance of complex queries in main memory database systems. In *ICDE*, pages 72–81, 1987.

[5] P. A. Boncz, M. Zukowski, and N. Nes. Monetdb/x100: Hyper-pipelining query execution. In *CIDR*, pages 225–237, 2005.

[6] V. Botea, D. Mallett, M. A. Nascimento, and J. Sander. PIST: an efficient and practical indexing technique for historical spatio-temporal point data. *GeoInformatica*, 12(2):143–168, 2008.

[7] V. P. Chakka, A. C. Everspaugh, and J. M. Patel. Indexing large trajectory data sets with SETI. In *CIDR*, 2003.

[8] P. Cudre-Mauroux, E. Wu, and S. Madden. Trajstore: An adaptive storage system for very large trajectory data sets. In *ICDE*, pages 109–120, 2010.

[9] D. Gawlick and D. Kinkade. Varieties of concurrency control in IMS/VS fast path. *DEB*, 8(2):3–10, 1985.

[10] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.

[11] S. Héman, M. Zukowski, N. J. Nes, L. Sidirourgos, and P. Boncz. Positional update handling in column stores. In *SIGMOD*, pages 543–554, 2010.

[12] M. G. Ivanova, M. L. Kersten, N. J. Nes, and R. A. Gonçalves. An architecture for recycling intermediates in a column-store. *TODS*, 35(4):24:1–24:43, 2010.

[13] J. Krueger, C. Kim, M. Grund, N. Satish, D. Schwalb, J. Chhugani, H. Plattner, P. Dubey, and A. Zeier. Fast updates on read-optimized databases using multi-core CPUs. *PVLDB*, 5(1):61–72, 2011.

[14] C. Lemke, K.-U. Sattler, F. Faerber, and A. Zeier. Speeding up queries in column stores. In *DaWaK*, pages 117–129, 2010.

[15] S. Manegold, P. Boncz, and M. L. Kersten. Generic database cost models for hierarchical memory systems. In *PVLDB*, pages 191–202, 2002.

[16] N. Meratnia and R. By. Spatiotemporal compression techniques for moving point objects. In *EDBT*, pages 765–782, 2004.

[17] D. Pfoser, C. S. Jensen, Y. Theodoridis, et al. Novel approaches to the indexing of moving object trajectories. In *Proceedings of VLDB*, pages 395–406, 2000.

[18] H. Plattner. A common database approach for OLTP and OLAP using an in-memory column database. In *SIGMOD*, pages 1–2, 2009.

[19] H. Plattner. SanssouciDb: An in-memory database for processing enterprise workloads. In *BTW*, volume 20, pages 2–21, 2011.

[20] J. Rao and K. A. Ross. Making B+- trees cache conscious in main memory. In *SIGMOD*, pages 475–486, 2000.

[21] S. Rasetic, J. Sander, J. Elding, and M. A. Nascimento. A trajectory splitting model for efficient spatio-temporal indexing. In *Proceedings of VLDB*, pages 934–945, 2005.

[22] C. S. J. Simonas Saltenis, S. T. Leutenegger, and M. A. Lopez. Indexing the positions of continuously moving objects. In *SIGMOD*, pages 331–342, 2000.

[23] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: a column-oriented DBMS. In *VLDB*, pages 553–564, 2005.

[24] H. Su, K. Zheng, H. Wang, J. Huang, and X. Zhou. Calibrating trajectory data for similarity-based analysis. In *SIGMOD*, pages 833–844, 2013.

[25] Y. Tao, D. Papadias, and J. Sun. The TPR*-tree: an optimized spatio-temporal access method for predictive queries. In *PVLDB*, pages 790–801, 2003.