Supporting Location-Based Services in a Main-Memory Database

Suprio Ray
Department of Computer Science
University of Toronto
Email: suprio@cs.toronto.edu

Rolando Blanco
SAP Canada, Waterloo
Email: rolando.blanco@sap.com

Anil K. Goel
SAP Canada, Waterloo
Email: anil.goel@sap.com

***Abstract*—With the proliferation of mobile devices and explosive growth of spatio-temporal data, Location-Based Services (LBS) have become an indispensable technology in our daily lives. The key characteristics of the LBS applications include a high rate of time-stamped location updates, and many concurrent historical, present and predictive queries. The commercial providers of LBS must support all three kinds of queries and address the high update rates. While they employ relational databases for this purpose, traditional databases are unable to cope with the growing demands of many LBS systems. Support for spatio-temporal indexes within these databases are limited to R-tree based approaches. Although a number of advanced spatio-temporal indexes have been proposed by the research community, only a few of them support historical queries. These indexing techniques, with support for historical queries, are unable to sustain high update and query throughput typical in LBS.**

Technological trends involving increasingly large main memory and core footprints offer opportunities to address some of these issues. We present several key ideas to support high performance commercial LBS by exploiting in-memory database techniques. Taking advantage of very large memory available in modern machines, our system maintains the location data and index for the past N days in memory. Older data and index are kept in disk. We propose an in-memory storage organization for high insert performance. We also introduce a novel spatio-temporal index that maintains partial temporal indexes in a versioned-grid structure. The partial temporal indexes are organized as compressed bitmaps. With extensive evaluation, we demonstrate that our system supports high insert and query throughputs and it outperforms the leading LBS system by a significant margin.

I. Introduction

The widespread adoption of GPS-enabled mobile devices and sensors have catapulted Location Based Services (LBS) to become a prominent technology. Geospatial Web services such as Google Maps also played a role in popularizing LBS applications. Started as an enabling technology for mobile asset tracking, LBS is the driving force behind a range of emerging applications such as location-based games and social networks, location-aware search and personalized advertising and weather services. The combination of factors, including the proliferation of mobile devices and the rise of novel applications, has led to the rapid growth of spatio-temporal data. Besides the data volume, LBS workloads also exhibit "velocity" of data. LBS applications are characterized by a very high rate of location updates and many concurrent location-oriented queries. These queries can be classified into "historical" or "past" queries, "now" or "present" queries and "predictive" or "future" queries. The commercial providers of LBS must support all three kinds of queries and deal with the high update rates. However, due to a rising customer base many of them are unable to handle the demands with the existing technology.

Commercial LBS offerings need to satisfy a diverse and often conflicting set of features. The customers would like to generate reports with different degrees of complexity. Some of these reports are based on present queries, or predictive range queries and others are long running historical reports based on past queries. With a large customer base the query workload in a given LBS system can be significant. At the same time the service providers must store and efficiently index location updates received at very high frequency. Šidlauskas et al. [21] noted that this rate can easily surpass 1 million or more location record updates per second. To address this complex set of requirements commercial LBS providers rely on relational databases to manage their data. To determine how traditional databases perform with update-intensive spatio-temporal workloads, we benchmarked [14] a database popular with the LBS industry. The database engine supports an R-tree based spatial index. We disabled the autocommit property and configured the buffer pool size to 64 GB so that the database completely fits in memory. The throughput achieved by the database was about 45K inserts/second, which is arguably low in this context. Our goal is to build database support for a full-fledged commercial LBS system. To this end, we intend to support millions of updates per second for one million moving objects, along with thousands of concurrent historical, present and predictive queries. This presents a few research challenges, including insert-optimized database storage and efficient spatio-temporal indexes.

The insert performance of traditional relational databases falls short, even when the database fits completely in memory. We argue that to meet the performance demands of Location-Based Services, it is essential to exploit recent advances in hardware and database technologies. Modern server-grade machines have large memory capacity, and some have hundreds of GB, even a few TB of memory. Each machine comes equipped with several tens or even hundreds of cores. Database technologies have evolved to take advantage of the abundant memory and processor capacity and in-memory databases are an active area of research. Several database vendors also have in-memory offerings, such as SAP HANA [15] and VoltDB [19]. When it comes to main-memory database storage design, there are a few options available as noted in Section II-A. However, we argue that it is possible to achieve better insert throughput than those from the in-memory row or columnar storage techniques. The location record updates are insert-only operations. There is no need to delete records from random locations in the table. A new record can always be inserted at the end of a table. To insert a new record in the table a new record id (RID) must be acquired, which is a unique identifier for each record. This can be performed without acquiring a lock, by incrementing the RID counter using an atomic CPU instruction. Also, location records are stored without any data compression, because typical compression techniques such as dictionary encoding are not that beneficial for position data. So, the associated cost of compressing records can be avoided. Furthermore, unlike regular transactions, the location updates do not need the strongest consistency guarantee. It may be considered okay even if a few location updates are missed.

Due to its importance in LBS applications, indexing moving objects has been an active area of research. Although a number of spatio-temporal indexes have been proposed [10], most of them are based on B-tree or R-tree and hence they suffer from the underlying limitations. These indexes are unable to sustain a very high rate of location data inserts/updates and the majority of these proposals deal with answering only present or predictive queries. Only a few of them can answer historical, present and predictive queries. A recently reported LBS system that supports historical and present queries is MOIST [6]. MOIST achieves 8K+ and 60K updates per second with one and 10 servers respectively for one million objects. Realizing the potential of memory based approaches, recently a parallel main-memory moving object indexing technique was proposed [21]. They achieved good update performance by exploiting efficient in-memory data structures. However, their system was designed to support present queries only, as at most 2 location records per object are maintained in memory. Hence it is not suitable as a real-world LBS system. To offer commercial LBS it is essential to maintain the history of each object and have support for historical queries.

We propose several key ideas that take advantage of the modern hardware resources. They are outlined below:

1) By taking advantage of large memory capacity we maintain the location table data and indexes for the past N (configurable) days in memory. Older data and indexes are maintained on disk. We also present an insert-efficient main-memory storage that is particularly suitable for LBS. Our storage model, *column fragment*, exploits insert parallelism by creating "multiple insertion points" that enable lockless insertion in each fragment.
2) We introduce a novel parallel in-memory spatio-temporal index called PASTIS (PArallel Spatio-Temporal Indexing System). PASTIS decomposes the spatial domain into grid cells and for each grid cell partial temporal indexes are maintained for moving objects that visited the cell. This makes it possible for separate threads to process the updates concurrently for different grid cells. The partial indexes are constructed as compressed bitmaps. Bitmaps offer memory efficiency and fast bitwise operations.
3) We present a fast bitmap compression technique. Although PASTIS can use any advanced bitmap compression technique such as that presented by Wu et al. [23], the features offered by our compressed bitmap such as fast random update and logical bit operations, are quite apt for our index.

With extensive evaluation studies we demonstrate that our system achieves excellent update throughput (millions of updates per second) and at the same time achieves very good query execution throughput (thousands of queries per second). For instance, our system achieves 3.2 million+ updates per second with 1 million moving objects. The update throughput of our system is significantly better than the reported throughput of MOIST and the relational database that we benchmarked.

The rest of the paper is organized as follows. Related works are mentioned in Section II. We describe the design considerations in Section III and system organization in Section IV. The details of main-memory storage for LBS and our spatio-temporal index, PASTIS, are in Sections V and VI respectively. Next we present our compressed bitmap in Section VII and a discussion in Section VIII. We describe the performance evaluation in Section IX and the conclusions are in Section X.

II. Related work

A. Main-memory storage

With the advent of large main-memory machines, there have been a number of projects addressing storage organization for main-memory databases. Row-store databases are typically utilized for update intensive workloads, whereas column-oriented databases are exploited for analytical workloads. SQL server's OLTP engine Hekaton [2] is an example of row-oriented main memory storage. IBM's Blink [13] is also a row-store main-memory database. MonetDB [16] is among the first column-store databases and it materializes intermediate results in main memory. To support updates with column-store usually a separate write-efficient store is maintained, besides a read storage. The main-memory column-store database C-store [16] uses a read-optimized store (RS) and writable store (WS), where all the inserts and updates are handled by WS. Data is moved from WS to RS as batch inserts by a tuple-mover. C-store maintains its data without any compression. SAP HANA is a commercial in-memory column oriented database that has a delta store for inserts and updates and a main store to support analytical queries [15]. Records in delta store are periodically merged into the main store. It uses dictionary encoding to compress the data. However, the movement of data or the merge process is a relatively time consuming operation. We explore the design space of main-memory storage for LBS in Section V and propose an insert-efficient main-memory storage called *column fragment*.

B. Spatio-temporal Index

The topic of spatio-temporal indexing has been addressed by a large body of research. These approaches can be classified based on whether they support indexing historical, present, predictive or combined past, present and future data. Nguyen-Dinh et al. provide a comprehensive survey of recent development in spatio-temporal access techniques [10]. The combined spatio-temporal access methods that have been proposed can be roughly classified into three categories: B-tree based, R-tree based and grid based.

BB^x-index [7] is a B-tree technique that extends B^x-tree [5]. B^x-tree converts the object locations into one dimensional space filling curve (SFC) codes and uses a B-tree to index these locations. BB^x-index maintains a forest of B^x-trees, each tree for a separate timestamp interval. Among the R-tree based techniques, HR-tree [9] is one of the earliest. It generates a new logical R-tree for each update, which leads to significant space usage and poor performance. The STR-tree [12] uses an R-tree to index moving object trajectories that are represented as connected line segments. However, this approach performs poorly when the trajectories are long, which would naturally occur as time progresses. A subsequent work, MV3R-tree [17] uses multiple versions of R-trees, in the same way BB^x-index does.

Since B-tree and R-tree are both external access methods, they are not scalable when it comes to supporting very high update rates. R-tree based approaches, in particular, perform poorly with update-heavy workloads due to the need to maintain the target structure constantly. B-tree based access methods on the other hand perform worse than those of R-trees for query-rich workloads, since R-trees can prune the search space much more efficiently than B-trees. With workloads involving millions of moving objects and thousands of simultaneous queries, neither can sustain high throughput.

Due to the availability of large main-memory machines, a few recent research projects focussed on in-memory indexing techniques. The MOVIES approach [3] enables concurrent queries and updates by letting the queries run on a read-only copy of the index structures, but this requires frequent rebuilding of short-lived data structures. Šidlauskas et al. [21] avoid this "stop the world problem" by using fine-grained concurrency control mechanisms. However, they only support present queries (not historical queries or predictive queries), whereas MOVIES only supports predictive queries.

MOIST [6] is a custom LBS system built over a key-value store. It employs traffic shedding and history data archiving. MD-HBase [11] is another LBS infrastructure that builds K-d tree and Quad tree indexes over a range-partitioned key-value store. It supports present range queries. Our indexing system supports all three types of range queries and at the same time offers better throughput than these approaches.

III. Design Considerations

We outline several key ideas to achieve the goal of handling high update rates and concurrently supporting past, present and predictive queries for LBS. We expand on the ideas introduced in Section I.

A. Main-memory storage for LBS

Many applications exhibit skew in the service requests for objects. A news website, for instance, receives significantly more page hits for breaking news than older news from the previous week. Such skew patterns can also be observed with the LBS workloads. More recent data from a vehicle fleet, for instance, is accessed much more often than older data. Wu and Madden [22] noted the significant request skew in LBS and reported that more than 50% of all queries accessed data from just the last day in their LBS system. Consequently, companies providing LBS typically maintain the past 30 to 45 days of data online in relational databases and older data is archived using database archiving techniques. Although the volume of location data even for 45 days is significant, it is feasible to keep the entire online data and the corresponding index in main memory. This is made possible by the availability of large main-memory machines, having hundreds of GB and even a few TBs. These observations led us to design our system as an in-memory database. We maintain in memory the past N days of data, and data older than that are kept on disk.

Besides the data volume, LBS workloads also exhibit high "velocity" due to frequent location updates from each moving object. Therefore, it is not enough to just provide in-memory storage capacity, but the storage must support record insertion at a high rate. To maintain history for each moving object the location records need to be stored in a database table. We assume that this *Location* table has the following schema: $\{$*ObjectId*,*Latitude*,*Longitude*,*Direction*,*Speed*,*Datestamp*$\}$

We propose an insert-efficient in-memory storage for the *Location* table. It is based on the idea of exploiting parallelism while inserting new location records in the table. For this, an in-memory column is partitioned into a number of segments called *column fragments*. Each column fragment is implemented as a dynamic array of primitive data types. The different column fragments from a table belonging in the same range of record identifiers are grouped as a *table fragment*. In Section V, we provide further details of our storage structure.

B. A novel parallel in-memory index

To answer past, present and predictive queries a high performance spatio-temporal index is necessary. The index must support high update rates. Traditional relational databases use R-tree based spatial index. These indexes inherit the underlying limitations. For instance, to add a new record in the index page level or R-tree node level locks must be acquired. This limits the multi-threaded scalability of these approaches.

We propose a novel in-memory spatio-temporal index we call PASTIS (PArallel Spatio-Temporal Indexing System). The main idea behind PASTIS is to decompose the spatial domain into grid cells and for each grid cell maintain partial temporal indexes for moving objects that visited the cell. The partial temporal indexes are constructed as compressed bitmaps. The compression of bitmaps provides memory storage efficiency. Bitmaps support fast random update operations. Moreover, the fast bitwise logical operations enabled by bitmaps allow us to attain good query performance. Our spatio-temporal index exploits multi-core parallelism by letting separate threads process the inserts/updates concurrently for different cells.

For an incoming location update u, a new record id *RID* is obtained and then the fields of the update record are appended into the corresponding column fragments. Updating the spatio-temporal index involves first computing the current grid cell id, then the corresponding temporal index data structures are updated. Further details of PASTIS can be found in Section VI.

C. Fast bitmap compression

PASTIS relies on compressed bitmaps to construct the partial temporal indexes. The performance of the underlying bitmap implementation is crucial to its update throughput and query performance. By taking advantage of hardware parallelism in bitwise CPU instructions, bitmaps offer superior alternative to other data structures such as hashtables or linked lists for membership determination. Compression is necessary, because bitmaps are not memory efficient. Since the compression and decompression can be compute intensive, techniques that can operate on compressed bitmaps are preferred. The Word Aligned Hybrid (WAH) compressed bitmap is an example of such approaches. WAH uses run-length encoding for compressing long sequences of 0's and 1's.

We introduce a bitmap compression algorithm that supports speedier bitmap operations than those in WAH. Our approach is based on the observation that instead of always using run-length encoding, it might be more compute efficient to use a bit array or id list for portions of the entire bitmap. In this, the bit range is split into fixed sized chunks. Depending on the density of each chunk a local decision is made as to which compression technique is to be used. Detailed description of our compressed bitmap is in Section VII.

IV. Overall system organization

Figure 1 shows the overall system organization. Our system is highly multi-threaded and takes advantage of the available processing cores. It uses three different thread-pools: to perform inserts into the Location table, to conduct updates into the index, and to execute queries. To reduce concurrency conflicts we use multiple record and index inserter queues and multiple query queues, such that each thread in a thread-pool manages a separate queue. Since the density of moving objects per grid cell changes continuously and is skewed, load-balancing is essential. This is discussed in Section VI-C

V. Insert-efficient main-memory storage

To support high location update rates in LBS it is imperative that the underlying storage for the Location table be optimized for record inserts. To achieve parallel performance with concurrent insertions in the Location table, we use the concept of "multiple insertion points". In this organization, each in-memory column is partitioned into a number of segments called column fragments. The entire record id (RID) space is subdivided such that each column fragment operates in a different RID space. The column fragments from different columns that belong in the same range of record identifiers constitute a table fragment. Each object is assigned to a table fragment so that all RIDs corresponding to the location updates for an object are monotonically incremental in its RID space.

Formally, let the entire RID space be $\{x|x = 1, 2, 3, ...N\}$, where N is the highest RID. Let there be y table fragments, where $y = 1, 2, 3, ..F$. Then the RID space R_y of table fragment y is the interval $[R * (y - 1) + 1, R * y]$, where $R = N/F$. Let there be M moving objects in the system. They are grouped into sets of objects $\{V_i|i = 1, 2, 3, ..F\}$, where each set V_i has $V = M/F$ objects. A function f is used to map each set V_i to a RID space R_y, where $f\colon V_i \to R_y$. When a new location update is received from a moving object $v_i \epsilon V_i$, it is always inserted at the end of its assigned table fragment. For instance, if v_i is mapped to R_y and r_y is the last assigned RID within R_y, a new RID is obtained by incrementing r_y by 1 before the new location update can be processed. A separate thread is dedicated to handle the inserts for a different table fragment, allowing concurrent updates into the same table without lock conflicts. To increment RID in a lockless manner, atomic instructions such as Compare-and-Swap or Fetch-and-Add can be used.

Other main-memory columnar storage models, such as C-store [16] or SAP HANA [15], maintain a separate writable storage, which must be periodically merged to the read-optimized store. This merge process can be expensive. In our model we have a single storage for the Location table that is used for both reads and writes. Since location data does not benefit from compression, we store the data "as is" and avoid the corresponding cost of compression using techniques such as dictionary encoding. We also do not perform any sorting on the columns of the Location table, unlike other column-stores. The new records from a particular object are always inserted at the end of the same table fragment, they are implicitly sorted by the timestamp column.

To evaluate the record insert performance of column fragment against row-store and column-store, we implemented memory resident column-store and row-store storages. Our column-store implementation resembles the L2-delta store of SAP HANA [15]. Records are inserted into the Location table as is, and no compression or sorting is performed. Prior to inserting a record, a new insert position or RID is acquired using the Fetch-and-Add atomic operation. We also tested atomic Compare-and-Swap (CAS), but found Fetch-and-Add to be faster. Our experiment involves inserting 1 billion location records with different numbers of inserter threads (2, 4, 8 and 16). The details of the dataset are in Section IX-A. To avoid any disk I/O, we preallocated enough memory to completely accommodate the entire dataset. The completion times of inserting 1 billion records with different numbers of threads are shown in Figure 2. As can be seen, the column fragment is 7X and 12X faster with two threads, and 34X and 39X faster with 16 threads than row and column stores respectively. This is due to the fact that with column fragments there is no contention over the atomic operation of acquiring a new RID. Also as each column fragment is managed by a dedicated thread there are fewer cache misses compared to the other two. We report the L2 cache misses observed during the insert operations in Figure 3. Note that the completion time with row store is better than that of the column store, because of fewer cache misses. The experiment demonstrates that column fragment is efficient for inserts. Row-store could also benefit from such fragmentation.

We adopt column fragment for the Location table because column-based storage is more efficient for analytical workloads. Currently, persistence of the Location table is provided by memory-mapped files. Location updates do not require the strongest consistency guarantee. Unlike transactional workloads, missing a few location records may be tolerated.

VI. PASTIS

In this Section we describe our spatio-temporal index, shown in Figure 4. First we describe the internal data structures. Then, we describe the update and query algorithms.

A. Index structure

The index is organized as a versioned grid in which the spatial domain is subdivided into regular sized grid cells. For a highly skewed location dataset it is also possible to structure the grid cells as quad-tree blocks, in which each block is recursively split into four blocks until they meet a criteria. PASTIS orders the grid cells using Z-order (also known as Morton-order) space filling curve. Therefore any location record can be positioned into a corresponding grid cell with its Z-order code. Each grid cell maintains partial temporal indexes for the objects that visited that cell. Whenever a location update *u* is received, the corresponding grid cell *SGrid*$_i$ (where i=1 to I, with I the total number of cells) is located and the partial temporal index structures are updated for that record.

A partial temporal index consists of an interval lookup table *Itab* with an entry for each time interval for the past *N* days. In each entry there is a compressed bitmap *CBmap* identifying the moving objects that were in the grid cell at the given time interval, and a hashmap *Hm-RIDList*. The hashmap associates each moving object with a list of record identifiers *RIDList*. Each identifier is used to locate in the Location table the actual records storing datestamp, latitude and longitude information for the object while at the grid cell during the time interval. The *RIDList* is implemented as a dynamic integer array. Time intervals are of *S* seconds (configurable value) for the past

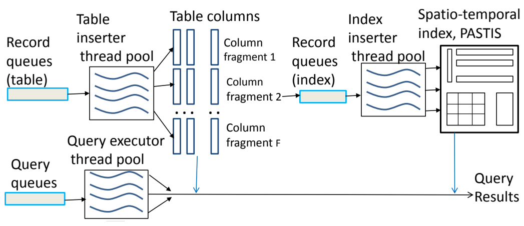

Fig. 1. **System organization**

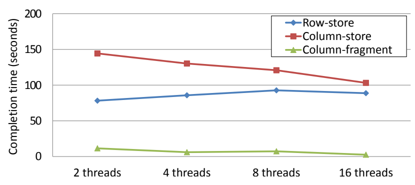

Fig. 2. **Completion time (1 billion record insert) with different storage organization**

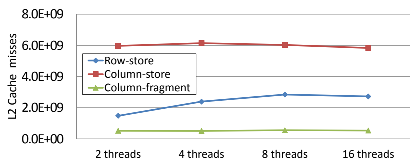

Fig. 3. **L2 cache misses with different storage organization**

configurable number of hours *H*. For data records older than *H* hours, the compressed bitmap *CBmap* and the hashmap *Hm-RIDList* are maintained for configurable intervals of time T (where $T = m * S$, for a configurable m). The bitmap is constructed by bitwise ORing the S seconds interval bitmaps, the hashmap by appending RID lists.

For a new update *u* with location *(x,y)*, datestamp ds_t, object id Obj_k and record id RID_l, let $SGrid_i$ be the determined grid cell and $Itab_i$ be the corresponding interval table. Here, ds_t is the datestamp at time *t*; *j*=1 to J, J being total intervals; k=1 to K, K being total objects and *l*=1 to L, L being the max record id. If ds_t maps to an existing interval TS_j in $Itab_i$, then the corresponding bitmap $CBmap_j$ and hashmap $Hm\text{-}RIDList_j$ are updated. Otherwise a new interval TS_j and related data structures are instantiated. For object id Obj_k, the *k*-th bit of the bitmap $CBmap_j$ is set. Furthermore, to make an entry for the new record id RID_l, the object id Obj_k is used to locate the RID list $RIDList_k$ within $Hm\text{-}RIDList_j$. Finally, RID_l is appended in the RID list of $RIDList_k$.

While processing a query, the checking of the bitmap *CBmap* to identify if an object was present during an interval is a fast operation. On the other hand, inspecting the *Hm-RIDList* is a relatively expensive operation; it involves a lookup in the hashmap *Hm-RIDList*, then the retrieval of the corresponding RID list *RIDList* and finally, obtaining the actual datestamps etc, from the table. Moreover, if the temporal index interval is small there will be fewer entries in the corresponding *Hm-RIDList* and the lists of record identifiers. Hence accessing a *Hm-RIDList* and corresponding *RIDList* pertaining to a smaller interval is faster than that for a longer interval. If a query interval completely overlaps a temporal index interval, there is no need to inspect the *Hm-RIDList*. For a query interval that partially overlaps an index interval, the *Hm-RIDList* is checked. Smaller temporal index intervals are more likely to be completely overlapped by a query interval. These are the considerations behind maintaining partial temporal indexes with finer granular interval *S* for more recent "hot" data and maintaining partial indexes with coarser granular interval *T* for "colder" data. Therefore, queries accessing the "hot" data need to inspect smaller intervals and hence can be faster. Partial index structures for data older than *N* days are kept in disk.

The partial index structures described above are used for answering historical and present queries. To efficiently support predictive queries (anticipated future locations) we keep a hashmap *PHm* keyed by moving object id (Obj_k), the value being a tuple (Loc_k) with the predicted latitude and longitude. An array of hashmaps *ArHm* associates a compressed bitmap $PBmap_i$ with each grid cell $SGrid_i$. The bitmap represents the predicted object status within the grid for a future configurable time interval F, where $T_F = T_{now} + \Delta t$. The anticipated future positions are interpolated using the model in [3]. For a location update with current location *(x,y)*, the predicted

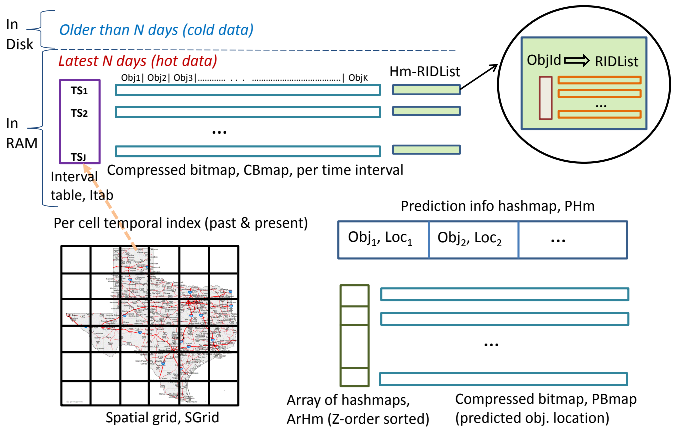
In Disk
Older than N days (cold data)
Latest N days (hot data)
Obj1 | Obj2 | Obj3 | ... | ObjK
Hm-RIDList
ObjId ⟹ RIDList
TS1
TS2
TSJ
...
In RAM
Interval table, Itab
Compressed bitmap, CBmap, per time interval
Prediction info hashmap, PHm
Per cell temporal index (past & present)
Obj1, Loc1
Obj2, Loc2
Spatial grid, SGrid
Array of hashmaps, ArHm (Z-order sorted)
Compressed bitmap, PBmap (predicted obj. location)

Fig. 4. **Structure of our spatio-temporal index, PASTIS**

```
Ensure: The record was inserted into table and RId is record id
Require: LocTable is database table, TemporalIdx is temporal index
 1: while RecordConcurrentQueue has more items do
 2:     {// First update predicted data structures}
 3:     LocnRecord ← RecordConcurrentQueue.pop()
 4:     OId ← LocnRecord.getObjId()
 5:     PredctdGCellId ← computePredctdGridCellId(LocnRecord)
 6:     PrevGCellId ← PredictivLookupDS.getPrevGCellId(OId)
 7:     if PrevGCellId != PredctdGCellId then
 8:         PredictivLookupDS.removeBitmapEntry(PrevGCellId,OId)
 9:         PredictivLookupDS.addBitmapEntry(PredctdGCellId,OId)
10:     PredictivLookupDS.insertIntoPHm(OId,PredictedLatLon)
11:     {// Next update past/present data structures}
12:     CurrGCellId ← computeCurrGCellId(LocnRecord)
13:     TemporalIdx ← SGridHashMap.find(CurrGCellId)
14:     TemporalIdx.insertLocnRecord(LocnRecord)
```

Fig. 5. **Algorithm Update**

position is calculated as $(x, y) + \vec{pv}.T_F$, (where $\vec{pv}$ is the projected velocity at time T_F) and entered in the hashmap *PHm*. Another prediction function, such as in Panda [4], could be used. If the anticipated grid cell id is different from current grid cell id, the array of hashmaps is used to locate and update the previously and currently predicted bitmaps.

B. Update processing

In Figure 5 we describe the update processing algorithm. For an incoming location update *u*, a new record id *RID* is obtained and then the fields of the update record *LocnRecord* are appended into the corresponding column fragments. Updating the index involves first updating the prediction data structures. In line 5 the predicted grid cell id is computed. If the previous grid cell entry for that object is different from the predicted, then the object id is removed from the bitmap corresponding to previously predicted grid cell and an entry is made into the bitmap for predicted grid cell (lines 7 to 9). Also, the Loc_k object is updated in the predictive hashmap *PHm* (line 10). The data structure *PredictivLookupDS* encapsulates both *PHm* and the array *ArHm* of hashmaps. Next, the data structures for the past and present are updated. The current cell id is computed

```
Require: LocnRecord is the location record to insert
 1: OId ← LocnRecord.getObjId()
 2: Datestamp ← LocnRecord.getDatestamp()
 3: RId ← LocnRecord.getRecordId()
 4: RecInterval ← computeInterval(Datestamp)
 5: CBmapHmRIDList ← this.Itab.find(RecInterval)
 6: if CBmapHmRIDList = NULL then
 7:     this.RWLock.lock()
 8:     Create a new CBmapHmRIDList instance
 9:     CBmapHmRIDList.updateObject(OId,RId)
10:     this.Itab.update(RecInterval,CBmapHmRIDList)
11:     this.RWLock.unlock()
12: else
13:     CBmapHmRIDList.updateObject(OId,RId)
```

Fig. 6. **Procedure TemporalIdx.insertLocnRecord**

```
Require: OId is the moving object id, RId is the record id
 1: RIDList ← this.HmRIDList.find(OId)
 2: this.RWLock.lock()
 3: if RIDList = NULL then
 4:     Create a new RIDList instance
 5: this.CBmap.add(OId)
 6: RIDList.add(RId)
 7: this.HmRIDList.insert(OId,RIDList)
 8: this.RWLock.unlock()
```

Fig. 7. **Procedure CBmapHmRIDList.updateObject**

and the corresponding temporal index structures are updated (line 14).

Figure 6 details the steps for updating the temporal index structures with the info from *LocnRecord*. First, the temporal interval is computed (line 4) and the corresponding bitmap *CBmap* and *Hm-RIDList* structures (together encapsulated by *CBmapHmRIDList*) updated (line 9 and 13). A lock is acquired only when a new instance of *CBmapHmRIDList* (lines 6 to 11) needs to be created. Updating the *CBmapHmRIDList* involves locating the RID list entry *RIDList* in the hashmap *HmRIDList* (Figure 7, line 1). Then a read-write lock is acquired, *CBmap* is updated and the RID for the location record is inserted into *RIDList* before releasing the lock (lines 2 to 8).

C. Load-balancing and skew handling

PASTIS is based on spatial partitioning of the spatial domain into regular grid cells. The distribution of the moving objects (the number of objects per cell) can be highly skewed and this distribution changes over time. Therefore, load-balancing is important for location update performance, especially with very high update rates. The goal of the load-balancing approach is to minimize the variation in the number of location records processed by the update processing threads. This can be done in two ways: either assign the partitions (cells) to threads using *Range assignment* or *Round-robin assignment*. Figure 8 shows the Round-robin assignment algorithm. Each index inserter thread manages a concurrent queue. In the Assignment step the grid cells are assigned to index inserter threads. A table inserter thread inserts a location record in the table fragment and then inserts it into the index inserter queue based on the cell the record belongs to and the cell to thread mapping. An index inserter thread simply processes the next record from its queue. A third load-balancing approach (we call *Adaptive*; Figure 9) does not do any cell assignment. Instead, a table inserter thread inserts a record into a randomly chosen queue. Each index inserter thread iterates over the queues and processes the next available record.

D. Query processing

In this Section we describe the past, present and predictive (future) query processing algorithms. The algorithm for historical range query is shown in Figure 10. The grid cells that are fully or partially covered by the query window *QRect* are computed in lines 2 and 3. We proceed to describe the steps for the partially covered cells. For each partially covered grid cell the corresponding temporal index is used to obtain the objects that were inside the cell during the interval (*EndDS* - *StartDS*) using bitwise OR (lines 8 - 10 of Figure 10, and Figure 11). This list of such objects is encoded in a partial result bitmap *tmpResltPartlyCovrd*. For each such moving object (i.e. for each bit that is set in *tmpResltPartlyCovrd*) inside every partially covered grid cell, the corresponding temporal index is utilized to inspect if that object was inside the query window *QRect* (line 18 of Figure 10). The details of this inspection is shown in Figures 12 and 13. This involves retrieving the *RIDList* for that object (line 1 of Figure 13) and for each RID entry, obtaining the corresponding location record fields of latitude, longitude and datestamp from the location table (lines 3 - 6). If the object location was inside the query rectangle and the datestamp with the query interval (line 8), the object is part of the past range query resultset.

```
Assignment:
 1: Assign grid cells to index inserter threads in a round-robin
    manner to produce the hashtable CellToThreadMappingHT
Table inserter thread:
 2: ...
 3: RecGridCellId ← computeCurrGridCellId(LocnRecord)
 4: threadId ← CellToThreadMappingHT.find(RecGridCellId)
 5: RecQueueArray[threadId].push(LocnRecord)
Index inserter thread:
 6: Initialize LocalRecQueue ← RecQueueArray[localThreadId]
 7: while true do
 8:     LocnRecord ← LocalRecQueue.pop()
 9:     Process LocnRecord and insert into index
```

Fig. 8. **Algorithm Round-robin assignment**

```
Table inserter thread:
 1: ...
 2: threadId ← RandomGenerator.nextRand()%NUM_THREADS
 3: RecQueueArray[threadId].push(LocnRecord)
Index inserter thread:
 4: for qIdx ← 0 to RecQueueArray.size() - 1 do
 5:     LocnRecord ← RecQueueArray[qIdx].pop()
 6:     Process LocnRecord and insert into index
```

Fig. 9. **Algorithm Adaptive**

The present range query algorithm utilizes the past range query algorithm (Figure 10), by specifying the end datestamp parameter to be "now" and start datestamp parameter to a few seconds earlier (a configurable value *offset*).

The predictive range query algorithm uses two main data structures: the prediction hashmap *PHm* and the array of hashmaps *ArHm* (associating the each grid cell to a prediction bitmap). The algorithm performs the following steps:

```
1. Compute the grid cells that are fully or partially covered by the
   query window QRect
2. For each fully or partially covered cell the intermediate result
   bitmap is obtained by performing a bitwise OR with the prediction
   bitmap
3. Initialize final result bitmap with fully covered result bitmap
4. For each bit that is set in partially covered result bitmap do
5.     Retrieve actual predicted latitude and longitude from PHm
6.     If the predicted location is inside QRect
7.         Set the bit representing the object in the final result bitmap
```

VII. Compressed Bitmap

The main idea behind our insert-efficient compressed bitmap is to split the bit range (each bit number identifying a moving object) into fixed size chunks. How bits are represented

```
Require: Reslt is result-set bitmap, LocTable is database table,
    STIdx is spatio-temporal index, QRect is query window, StartDS
    and EndDS are query interval datestamps
 1: {//Local variables: tmpResltFullyCovrd, tmpResltPartlyCovrd }
 2: FullyCovrdCells ← getFullyCoveredCells(QRect)
 3: PartlyCovrdCells ← getPartiallyCoveredCells(QRect)
 4: for gridCell in FullyCovrdCells do
 5:     TemporalIdx ← STIdx.getTemporalIdxAt(gridCell.id)
 6:     TemporalIdx.getIntervalMatchingObjs(tmpResltFullyCovrd,
        StartDS,EndDS)
 7:     Reslt ← BitwiseOr(Reslt, tmpResltFullyCovrd)
 8: for gridCell in PartlyCovrdCells do
 9:     TemporalIdx ← STIdx.getTemporalIdxAt(gridCell.id)
10:     TemporalIdx.getIntervalMatchingObjs(tmpResltPartlyCovrd,
        StartDS,EndDS)
11: tmpResltPartlyCovrd.BitwiseMinus(tmpResltFullyCovrd)
12: NumBitsSet ← tmpResltPartlyCovrd.NumBitsSet()
13: BitmapSetIterator ← tmpResltPartlyCovrd.getIterator()
14: for gridCell in PartlyCovrdCells do
15:     TemporalIdx ← STIdx.getTemporalIdxAt(gridCell.id)
16:     BitmapSetIterator.ReSet()
17:     for (nxtBit ← BitmapSetIterator.GetnxtBitSet()) != NULL do
18:         if Reslt.IsSet(nxtBit) != TRUE then
19:             if (TemporalIdx.isObjInGrdCellAndIntrvl(nxtBit,
                QRect,StartDS,EndDS,LocTable)) then
20:                 Reslt.SetBit(nxtBit)
```

Fig. 10. **Algorithm PastRangeQuery**

```
Require: tempReslt is temporary result-set bitmap, StartDS and
    EndDS are query interval datestamps, LocTable is database table
 1: IntervalsCovrd ← getCoveredIntervals(StartDS,EndDS)
 2: for Interval in IntervalsCovrd do
 3:     CBmapHmRIDList ← this.Itab.find(Interval)
 4:     PartialRsltBitmap ← CBmapHmRIDList.getObjsBitmap()
 5:     this.ReadWriteLock.ReadLock()
 6:     tempReslt ← BitwiseOr(tempReslt, PartialRsltBitmap)
 7:     this.ReadWriteLock.UnLock()
```

Fig. 11. **Procedure TemporalIdx.getIntervalMatchingObjs**

```
Require: QRect, StartDS, EndDS, LocTable same as before
 1: IntervalsCovrd ← getCoveredIntervals(StartDS,EndDS)
 2: for Interval in IntervalsCovrd do
 3:     CBmapHmRIDList ← this.Itab.find(Interval)
 4:     IsInside ← CBmapHmRIDList.isObjInGrdCellAndIntrvl(
        ObjId, QRect,StartDS,EndDS,LocTable)
```

Fig. 12. **Procedure TemporalIdx.isObjInGrdCellAndIntrvl**

in each chunk depends on the bit density within the chunk:

- Run-length encoding is used if a contiguous sequence of bits is set, and there is only one contiguous sequence within the chunk. We refer to these chunks as *contseq* (for "contiguous sequence") chunks.
- The chunk is represented as an array of bit numbers if less than a configurable number of bits is set. Although *objlist* (for object list) would be more appropriate, we refer to these chunks as *ridlist* chunks.
- In any other case the chunk is represented as a bit vector. We refer to these chunks as *bitvect* chunks.

A bitmap is a list of chunks with the following properties:

- The state of each bit cannot be determined by more than one chunk (i.e., all chunks are disjoint).
- If a chunk C_i occurs before chunk C_k, then the first bit number in C_i's range is before the first bit in C_k's range.
- Each chunk contains at least one bit set.

The requirement for a chunk to represent the state of a fixed number of bits is relaxed in the case of contseq chunks. Contseq chunks can determine the state for a larger range of bits, but this range must be a multiple of the range size for the other types of chunks. For each bitmap two fields are kept, *FirstChunk* and *LastChunk* pointing to the first and last chunks in the list of chunks. Each chunk has an encoding field,

```
Require: ObjId, QRect, StartDS, EndDS, LocTable same as before
 1: RIDList ← this.HmRIDList.find(OId)
 2: for r=0 to ObjId.size()-1 do
 3:     Rid ← RIDList[r]
 4:     Lat ← LocTable.getLatitude(Rid)
 5:     Lon ← LocTable.getLongitude(Rid)
 6:     Datestamp ← LocTable.getDatestamp(Rid)
 7:     ReturnVal ← FALSE
 8:     if Datestamp ≥ StartDS AND Datestamp ≤ EndDS AND
        QRect.Contains(Lat, Lon) then
 9:         ReturnVal ← TRUE
10:         return
```

Fig. 13. **Proc CBmapHmRIDList.isObjInGrdCellAndIntrvl**

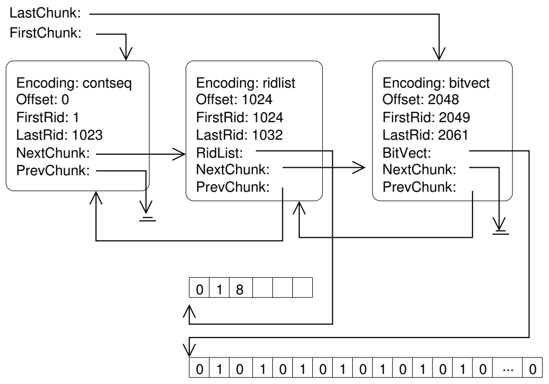

Fig. 14. **Bitmap Implementation**

specifying the type of chunk, and an offset field implemented as a 64 bit unsigned integer. Two fields *FirstRid* and *LastRid* determine the first and last bits on in each chunk. An example of our bitmap is shown in Figure 14.

A. Bitmap transformations

As entries are added or removed from bitmaps it may be beneficial to change how bits are specified in one or multiple chunks, if the new chunk representation uses less memory than the previous representation. Two chunk transformation operations, namely *TransformOnAdd* and *TransformOnRemove*, evaluate and possibly transform the representation of a chunk.

TransformOnAdd is invoked when new entries are being added. The goal is to identify whether the bitvect or ridlist can be changed to a contseq representation, which requires less memory than both. The steps of TransformOnAdd as follows:

```
1. If chunk-type is ridlist or bitvect
2.     If number-of-entries-in-chunk = (LastRid - FirstRid + 1)
3.         Reset chunk's encoding to contseq
```

TransformOnRemove operation is invoked when unsetting bits for bitvect chunks and the goal is to change the representation to a ridlist if the number of bits set in the bitvect is less than the capacity of a ridlist. The steps of TransformOnRemove algorithm are shown below:

```
1. If chunk-type is bitvect
2.     If number-of-entries-in-chunk ≤ capacity-of-ridlist
3.         Reset chunk's encoding to ridlist
```

When a chunk becomes a contseq chunk a check is performed to see if we can *coalesce* it either with the previous or next chunk. Coalescing occurs only for contseq chunks and it involves replacing two chunks with a single chunk.

VIII. Discussion

PASTIS is essentially a versioned grid organization that maintains for each time interval the partial trajectory of each moving object. PASTIS utilizes insert-efficient compressed bitmap and dynamic integer array (RID list) to maintain the

TABLE I. TRACE FILE DETAILS

Dataset name	**Num. of mobile objects**	**Num. of location records**	**Size on disk**
10 million	10,000	10,000,000	445 MB
100 million	100,000	100,000,000	4.5 GB
1 billion	1,000,000	1,000,000,000	46 GB

partial trajectory. It could be argued that this can also be implemented with an in-memory R-tree based approach. To evaluate the performance of such an approach we replaced the *CBmap* and *Hm-RIDList* structures with an efficient in-memory R*-tree. An in-memory R*-tree is used to track the partial trajectory of all the moving objects during a time interval for each grid cell. A location update is processed by the R*-tree as a combination of deletion, updating the trajectory MBR and re-insertion operations.

A location update involves inserting a new record in the table and updating the index. Figure 15 shows the breakdown of the completion time to perform 10 million location updates and contrasts PASTIS implemented with compressed bitmap and RID list vs. in-memory R*-tree. The data was obtained by Callgrind profiler. The R*-tree approach takes 10X more time. Also, whereas the *Update Temporal index* step takes 60.4% of overall time with the bitmap approach, it takes 96% of the overall time with the R-tree approach. Note that *Insert into table* step takes only a small fraction of overall time due to column fragment storage.

Since concurrency is a key performance bottleneck, our system exploits several high performance concurrency control features. The shared counter variables were implemented with atomic Fetch-and-Add. Efforts were made to avoid the usage of locks. However, in a few cases where it was unavoidable we used high performance spin locks. It was shown that [21] spin locks provided the best performance compared to other options, especially when the locked objects are held for relatively short period of time. We used two concurrent collection classes: concurrent hashmap and concurrent blocking queue from the Intel TBB library.

IX. PERFORMANCE STUDIES

A. Experimental Setup

The performance studies were conducted on a HP Z820 machine with 256 GB of RAM and 2 Intel Xeon E5-2670 processors having a total of 16 cores (32 h/w threads), each running at 1200 MHz. The OS was Suse Linux SLES 11.1.

The polyline shapefiles of Texas from the TIGER dataset [18] were fed into the mobility trace generator MOTO [8] to generate the traces. We modified MOTO to generate multiple trace files, each file for a different table fragment. Mobility traces were generated for each object for 1000 timestamps (equivalent to 10,000 seconds). We generated the trace files for 3 different data sizes, as shown in Table I. Table II shows various configuration parameter settings.

B. Update Performance

We first evaluate the location update performance (insertion into the Location table and updating the index). In our experimental setup we utilized 10 dedicated threads to handle inserts into the Location table for 10 different column fragments.

1) Load-balancing algorithms: To determine the impact of the load-balancing approaches on location update performance we implemented the algorithms mentioned in Section VI-C, namely, Range assignment, Round-robin assignment and Adaptive. The throughput achieved by them with 1 billion location updates is shown in Figure 16. The Adaptive algorithm consistently achieved better throughput than the other two approaches as the number of *index inserter threads* are varied. To determine the reason for this, we plot the standard deviation of the number of records processed by the threads in Figure 17. The standard deviation was the least for the Adaptive algorithm. It is expected that as the number of threads are increased the variation among them is reduced. This can be clearly observed for Range assignment. With Round-robin, this variation was lower than that of Range assignment, but still higher than Adaptive. Note that Adaptive achieved over 3.2 million updates per second and this throughput became steady after 16 to 20 threads. Part of the reason is that 10 additional threads are dedicated as *table inserters*. We use Adaptive as the load-balancing algorithm in all subsequent experiments.

TABLE II. PARAMETER SETTINGS

Parameter	**Settings**
Space domain	Texas, 1251 km x 1183 km
Num. of road segments	56832846
Time duration, *timestamps*	1000
Update frequency, *seconds*	10
Updates (num. of records)	10 million, 100 million, **1 billion**
Range query area, km^2	1 , **4**, 16
Range query interval, %	1.5, 3, **6**, 12

2) Number of spatial partitions (grid resolution): To observe the effect of the number of partitions on location update performance, we vary the number of partitions (grid cells). Figure 18 shows the throughput achieved with the 3 datasets as the number of partitions are increased from 256 to 4096. The number of index inserter threads were fixed at 20. As can be seen, the throughput remains steady regardless of the number of partitions for all three datasets. This suggests that the Adaptive algorithm does a good job of load-balancing and grid resolution has no significant impact on update throughput. For the remaining experiments we use 1024 partitions.

3) Temporal index interval length: An index interval length is the duration of time interval for which all location records in a particular grid cell are recorded in the corresponding temporal index. The smaller the index interval length, the more temporal index instances are created. Intuitively, it would require more computation to process more temporal index instances. To evaluate the throughput at different index interval lengths, we vary the length to 2, 4, 6, 8 and 10 minutes. Figure 19 shows the corresponding throughputs. The throughput is increased as the interval length is increased, for instance, it is 3.5 million updates per second for dataset - 1 billion with interval length 4. This is a good trend, because in a real-world LBS application this interval length S for the records received within the last H (where, H = 24 hours) would be higher than 2 minutes, such as 10 to 30 minutes. For records older than 24 hours, this length T could be much higher, such as a few hours. Unless otherwise specified, we use 2 minute as the default.

4) Storage requirements: The storage requirement is a key consideration when its comes to adopting an index solution. To this end, we report the total size of all bitmaps created by our index by varying the temporal index interval lengths, to 2, 4, 6, 8 and 10 minutes. As shown in Figure 20, the total size of all the bitmaps is quite small, and is dependent on the temporal index interval length. As noted previously, in a real-world LBS

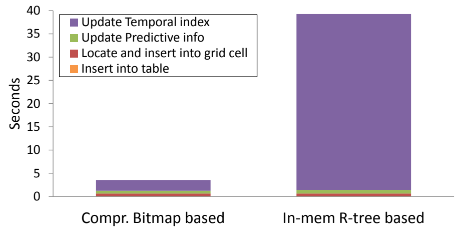

Fig. 15. **Breakdown of time for location update**

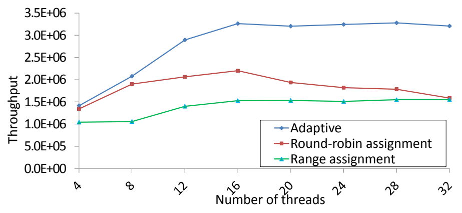

Fig. 16. **Update throughput with different load-balancing approaches**

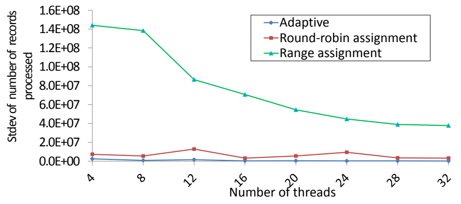

Fig. 17. **Standard dev. of number of records processed with different load-balancing approaches**

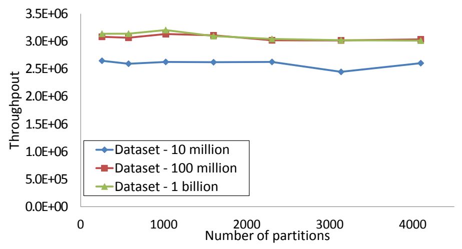

Fig. 18. **Update throughput with different number of partitions**

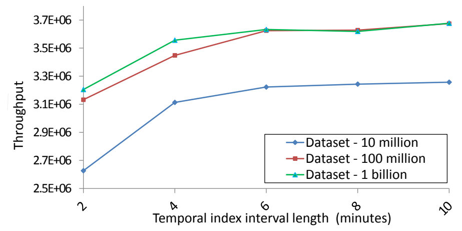

Fig. 19. **Update throughput with different temporal index interval length**

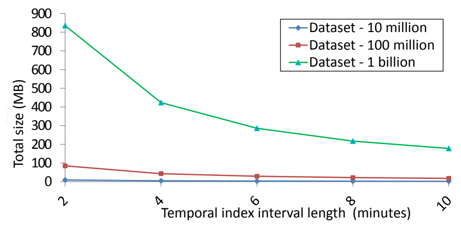

Fig. 20. **Total size of all bitmaps with different temporal index interval length**

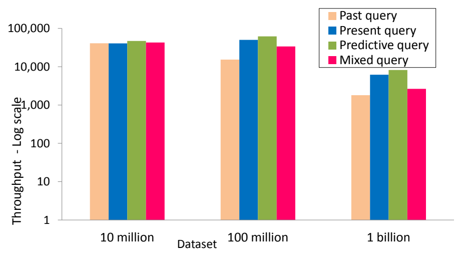

Fig. 21. **Query throughput**

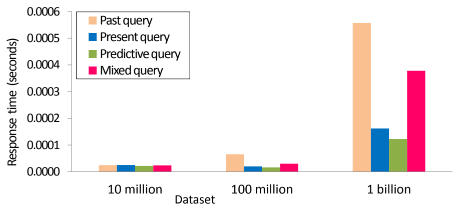

Fig. 22. **Average query response time**

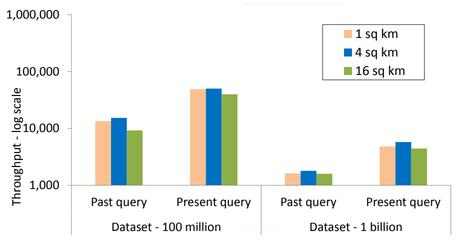

Fig. 23. **Query throughput - spatial extent**

applications the temporal index interval length would be much larger, and hence the storage requirements would be lower.

C. Query Performance

In this Section we evaluate concurrent range query performance. All queries were executed simultaneously with record updates. The query execution is started after at least 5% of the dataset is populated. Four types of range queries were evaluated: past (historical), present, predictive (future), and mixed. The mixed queries were generated by issuing past, present and predictive queries at equal ratio 1:1:1. Since LBS applications are dominated by updates, rather than queries, we assign the available threads in 2:1 ratio to update processing as opposed to query processing. We used update to query ratio 1000:1, so that one query was issued for every 1000 updates. No significant decrease in update throughput was observed.

Figure 21 shows the throughputs of past, present, predictive and mixed queries. For all three datasets, the system achieved throughputs of thousands of queries per second. The average query response times are shown in Figure 22. As expected, past query response times are the longest. However, even with the dataset - 1 billion, the past query response time was less than a millisecond. For present, predictive and mixed queries the response times were lower than those of past queries. This shows that in-memory bitmaps can help accelerate query performance.

1) Spatial extent: A key parameter, when generating queries, is the spatial extent or the spatial dimension of the query window. We used three spatial extent values: 1 sq. km, 4 sq. km and 16 sq. km [21]. The Figure 23 compares the throughput of past and present queries for the 3 different spatial dimension sizes. For both the queries the query throughput is the lowest when the spatial extent is 16 sq. km. This is expected because when the dimension is larger, there are more moving objects and correspondingly, more records to be processed. Interestingly, the throughput is slightly lower at 1 sq. km than at 4 sq. km. To explain this, note that if a grid cell is partially overlapped by the query window it is more expensive to process than if it is fully covered. A cell is more likely to be partially covered at 1 sq. km than at 4 sq. km.

2) Interval length: The temporal length of the queries, expressed as a fraction of the recorded history [17], is an important parameter for the past queries. We used 4 query interval length values to evaluate the performance of the past queries: 1.5%, 3%, 6% and 12%. The Figure 24 compares the throughput of past range queries for the 4 different interval lengths and datasets 100 million and 1 billion. In all cases, the throughput is the best at interval length 1.5%. The throughput is decreased as the interval length is increased.

D. Comparison with other indexes and other LBS systems

Spade [1] is a popular benchmark that compares a number of spatio-temporal indexes. We evaluate two representative spatio-temporal indexes, B^x-tree and TPR-tree [20], from Spade using a setup described in Section IX-A. Like other in-memory databases, such as SAP HANA, in our system the Location table is persisted, and the index is built when the table is loaded into memory. To do a fair comparison, we modified the Spade benchmark code to make the B^x-tree and TPR-tree indexes memory resident and not persist into disk. Figure 25 compares their update and query throughputs of B^x-tree and TPR-tree against PASTIS for the 10 million record dataset. For both update and query, our approach shows orders of magnitude better throughput than B^x-tree and TPR-tree.

A paper on LBS system MOIST reported [6] that it attained

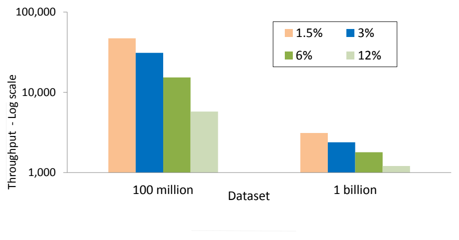

Fig. 24. **Query throughput - effect of interval length**

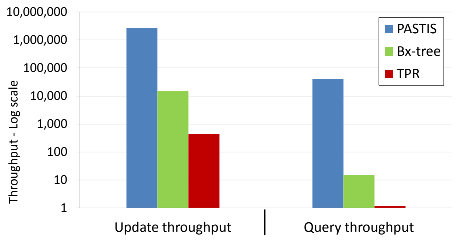

Fig. 25. **PASTIS vs B^x-tree vs TPR**

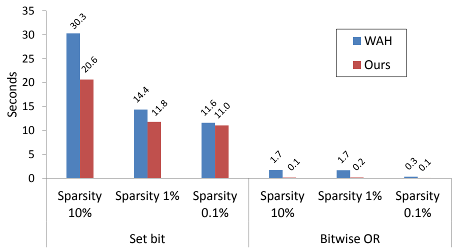

Fig. 26. **WAH vs our compressed bitmap**

8000+ updates per second and 60K updates per second with 1 server and 10 servers respectively for one million moving objects. Although a direct comparison is not possible, we demonstrated in Figure 18 that with a single server we achieve over 3.2 million updates per second for one million objects. This is a speedup of 400X over MOIST for a single server. MD-HBase [11] reported a peak throughput of 220K location updates per second (in a 16-node cluster), which is one order of magnitude lower than ours. They noted their best present range query average response time to be about 500 milliseconds. Our past and present range query response times are less than a millisecond for all 3 datasets. Most commercial LBS providers use traditional relational databases for data management. As reported in [14], a popular database achieves only about 45K updates per second even when it fits in memory.

E. *Compressed bitmap performance*

To compare the performance of our compressed bitmap against the state-of-the-art WAH compression, we create 1000 compressed bitmaps, each representing a partial temporal index with 1 million moving objects. We randomly set the bits of the bitmaps with 3 different sparsity: 10%, 1% and 0.1%. The sparsity determines how many bits are set, for instance, with 10% sparsity 1 out of every 10 bits are set randomly. We report the completion time to set the bits of all 1000 bitmaps for a particular sparsity. Since fast bitwise OR operation is important for query performance we also report the completion time to perform logical OR operation with the 1000 bitmaps. Figure 26 shows that our compressed bitmap is faster than WAH with both set bit and bitwise OR operations for all sparsity levels.

X. Conclusions

Location-Based Services have come to play important roles in various facets of our lives. With the exponential growth of spatio-temporal data, many commercial LBS systems are unable to meet the growing customer demands. It is necessary that the databases, on which LBS providers depend, offer sufficient performance to handle very high update rates, while supporting many concurrent past, present and future queries.

To address these requirements, we propose a combination of in-memory based techniques. We present an insert-efficient main-memory storage for LBS. We introduce PASTIS, a parallel in-memory spatio-temporal index that supports historical (past), present and predictive (future) queries. PASTIS is a versioned grid organization that utilizes compressed bitmaps to maintain partial temporal indexes for objects that visited each grid cell in a 2D spatial domain. By completely maintaining the data and index for the past active N days in memory, our system avoids any disk access latency for updates and queries. Thread-level parallelism and fine-grained concurrency control, supported by fast bitmap operations help our system to achieve high update and query performance. With extensive evaluations we demonstrate the superior performance of our system over existing approaches.

References

[1] S. Chen, C. S. Jensen, and D. Lin. A Benchmark for Evaluating Moving Object Indexes. *PVLDB*, 2008.

[2] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: Sql server's memory-optimized oltp engine. In *SIGMOD*, pages 1243–1254, 2013.

[3] J. Dittrich, L. Blunschi, and M. A. V. Salles. Indexing Moving Objects Using Short-Lived Throwaway Indexes. In *SSTD*, pages 189–207, 2009.

[4] A. M. Hendawi and M. F. Mokbel. Panda: A Predictive Spatio-temporal Query Processor. In *SIGSPATIAL GIS*, pages 13–22, 2012.

[5] C. S. Jensen, D. Lin, and B. C. Ooi. Query and update efficient B+-tree based indexing of moving objects. In *VLDB*, pages 768–779, 2004.

[6] J. Jiang, H. Bao, E. Y. Chang, and Y. Li. MOIST: A scalable and parallel moving object indexer with school tracking. In *PVLDB*, 2012.

[7] D. Lin, C. S. Jensen, B. C. Ooi, and S. Šaltenis. Efficient indexing of the historical, present, and future positions of moving objects. In *MDM*, pages 59–66, 2005.

[8] MOTO (Moving Objects Trace generatOr). http://moto.sourceforge.net/.

[9] M. A. Nascimento and J. R. O. Silva. Towards historical R-trees. In *SAC*, pages 235–240, 1998.

[10] L.-V. Nguyen-Dinh, W. G. Aref, and M. F. Mokbel. Spatio-Temporal Access Methods: Part 2 (2003 - 2010). *IEEE Data(base) Engineering Bulletin*, 33:46–55, 2010.

[11] S. Nishimura, S. Das, D. Agrawal, and A. E. Abbadi. MD-HBase: A Scalable Multi-dimensional Data Infrastructure for Location Aware Services. In *MDM*, 2011.

[12] D. Pfoser, C. S. Jensen, and Y. Theodoridis. Novel Approaches in Query Processing for Moving Object Trajectories. In *VLDB*, 2000.

[13] V. Raman, G. Swart, L. Qiao, F. Reiss, V. Dialani, D. Kossmann, I. Narang, and R. Sidle. Constant-Time Query Processing. In *ICDE*, pages 60–69, 2008.

[14] S. Ray, R. Blanco, and A. K. Goel. Enhanced Database Support for Location-Based Services. In *Intl. Workshop on GeoStreaming*, 2013.

[15] V. Sikka, F. Färber, W. Lehner, S. K. Cha, T. Peh, and C. Bornhövd. Efficient transaction processing in SAP HANA database: the end of a column store myth. In *SIGMOD*, 2012.

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

[17] Y. Tao and D. Papadias. MV3R-Tree: A Spatio-Temporal Access Method for Timestamp and Interval Queries. In *VLDB*, 2001.

[18] TIGER®. http://www.census.gov/geo/www/tiger.

[19] VoltDB. http://voltdb.com/.

[20] S. Šaltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the Positions of Continuously Moving Objects. *Sigmod Record*, 2000.

[21] D. Šidlauskas, S. Šaltenis, and C. S. Jensen. Parallel Main-Memory Indexing for Moving-Object Query and Update Workloads. In *SIGMOD*, pages 37–48, 2012.

[22] E. Wu and S. Madden. Partitioning techniques for fine-grained indexing. In *ICDE*, pages 1127–1138, 2011.

[23] K. Wu, E. J. Otoo, and A. Shoshani. Compressing Bitmap Indexes for Faster Search Operations. In *SSDBM*, 2002.