U²SOD-DB: A Database System to Manage Large-Scale Ubiquitous Urban Sensing Origin-Destination Data

Jianting Zhang
Department of Computer Science
City College, the City University of New York
New York, NY, 10031
jzhang@cs.ccny.cuny.edu

Hongmian Gong
Department of Geography
Hunter College, the City University of New York
New York, NY, 10065
gong@hunter.cuny.edu

Camille Kamga
Department of Civil Engineering
City College, the City University of New York
New York, NY, 10031
ckamga@utrc2.org

Le Gruenwald
School of Computer Science
University of Oklahoma
Norman, OK 73071
ggruenwald@ou.edu

ABSTRACT

Volumes of urban sensing data captured by consumer electronic devices are growing exponentially and current disk-resident database systems are becoming increasingly incapable of handling such large-scale data efficiently. In this paper, we report our design and implementation of U²SOD-DB, a column-oriented, Graphics Processing Unit (GPU)-accelerated, in-memory data management system targeted at large-scale ubiquitous urban sensing origin-destination data. Experiment results show that U²SOD-DB is capable of handling hundreds of millions of taxi-trip records with GPS recorded pickup and drop-off locations and times efficiently. Spatial and temporal aggregations on 150 million pickup locations and times in middle-town and downtown Manhattan areas in the New York City (NYC) can be completed in a fraction of a second. This is 10-30X faster than a serial CPU implementation due to GPU accelerations. Spatially joining the 150 million taxi pickup locations with 43 thousand polygons in identifying trip purposes has reduced the runtime from 30.5 hours to around 1,000 seconds and achieved a two orders (100X) speedup using a hybrid CPU-GPU approach.

Categories and Subject Descriptors

H.2.8 [Database Management]: Database applications – Data mining, Spatial databases and GIS

General Terms

Management, Design

Keywords

Urban Sensing, Origin-Destination, High-Performance, GPGPU, Spatial Aggregation, Temporal Aggregation, Spatial Join

1. INTRODUCTION

With the increasing availability of imaging, locating and other types of sensing technologies on portable wireless devices, huge amounts of pervasive urban sensing data are being captured at ever growing rates. For example, the approximately 13,000 taxicabs in the New York City (NYC) equipped with GPS devices generate more than half a million taxi trip records per day. Cell phone call logs represent a category of data at an even larger scale [1, 2]. Also as visitors travel around the world more frequently, location-dependent social networks such as FourSquares [3], and, location-enhanced social media such as text posted to wiki sites [4], images and videos posted to Flickers and YouTube [5], can also potentially generate large-amounts of spatial-temporal data. All the three types of data have a few features in common: (1) they are produced and collected by end users using commodity sensing devices and are rich in data volumes in urban areas; and (2) they are a special type of spatial-temporal data with an origin location and a destination location in geo-referenced space domain and a starting time and an ending time in the time domain. However, the intermediate locations between origins and destinations are either unavailable, inaccessible or unimportant. Compared with traditional geographical data collected by government agencies for urban planning and city administration purposes, these data can be more effective to help people understand the real dynamic of urban areas with respect to spatial/temporal resolutions and representativeness. We term such data as Ubiquitous Urban Sensing Origin-Destination data, or U²SOD data, for notation convenience. Despite the close relationships between U²SOD data and the Spatial Databases (SDB) [14] and Moving Object Databases (MOD) [6, 7], our experiences have shown that traditional disk-resident and tuple/row oriented spatial databases and moving object databases are ineffective in processing large-scale U²SOD data for practical applications.

Towards this end, we have designed a new data management system, termed U²SOD-DB, to address the technical challenges that we have encountered when processing about 300 million taxi trip records during an approximate two years period in NYC. U²SOD-DB is an in-memory database system and adopts a time-segmented column-oriented data layout strategy. A set of high-performance spatial and temporal indexing and query processing techniques has been developed to efficiently aggregate and join U²SOD data and their auxiliary geographical data (e.g., urban infrastructure) to understand urban dynamics. Parallel accelerations using General Purpose computing on Graphics Processing Units (GPGPU) technologies [8] are incorporated where appropriate. The rest of the paper is arranged as the follows. Section 2 introduces background and related work. Section 3 presents the system architecture and implementation details of the key components. Section 4 reports the performance evaluations. Finally, Section 5 concludes the paper and outlines the future work.

UrbComp'12, August 12, 2012. Beijing, China.

2. BACKGROUND, MOTIVATIONS AND RELATED WORK

The increasingly available location data generated by consumer wireless portable devices, such as GPS, GPS enhanced cameras and GPS/WiFi/Cellular enhanced mobile phones, have significantly changed the ways of collecting, analyzing, disseminating and utilizing urban sensing data. Traditionally city government agencies are responsible for collecting various types of geographical data for city management purposes, such as urban planning and traffic control. The data collection is usually done through sampling, typically at coarse-resolutions, and questionnaire-based investigations which often incur long turn-around periods. In contrast, as consumer mobile devices become ubiquitous, similar data obtained from GPS-traces and mobile phone call logs, has much finer resolutions. With the help of privacy and security related technologies, the aggregated records from such ubiquitous urban sensing data can be enormously helpful in understanding and addressing a variety of urban related issues. Research groups from both academia (e.g., MIT Sensible City Lab) and industries (e.g., IBM Smart Plant Initiative and Microsoft Research Asia) have developed quite a few techniques to utilize such data and understand the interactions among people and their location/mobility at both the city level (e.g., Beijing [9], Boston [10] and Rome [1]), social group level (e.g., friends [11] and taxi-passenger pairs [12]) and the individuals level [13]. However, most of the existing studies focus on the data mining aspects of such ubiquitous urban sensing data through case studies while largely leaving the data management aspects untouched. Lacking proper data management techniques can result in signficant technical hurdles to make full use of such data for societal impacts.

Although Geographical Information Systems (GIS) and Spatial Databases (SDB) [14] are the commonly used tools to handle geo-referenced spatial data, we argue that existing GIS and SDB technologies and available tools are inefficient and/or insufficient in managing large-scale, location dependent and time-varying ubiquitous urban sensing data. First, when looking back through the history, GIS and SDB technologies are mostly driven by the needs of managing cadastral types of geographical data more than five decades ago which are quite different from the ubiquitous urban sensing data we have today. Second, although GPS trajectories can be naturally modeled as moving object data and existing moving object databases technologies for indexing and query processing (e.g., [6, 7, 26]) can be exploited, many other types of ubiquitous urban sensing data can not be efficiently managed by existing databases. Third, existing commercial and open source GIS and SDB are slow in adopting hardware acceleration techniques. Existing disk-resident GIS and SDB software tools are mostly designed for online transactions rather than analytics and they can be orders faster if they are tailored for read-only data [15, 16]. Additional signficant speedup is possible if modern hardware acceleration techniques are exploited, e.g., flash memory for off-chip data access, CPU caches and GPU shared memory for on-chip data access, and, multi-core CPUs and many-core GPUs for parallel computing [17]. Our own experiences in processing 300 millions taxi trip records in NYC have shown that the performance of existing GIS and SDB software tools is unacceptable when processing urban sensing data at this scale – a simple spatial join may take dozens of days even on a current high-end server. This may also explain why most of existing studies that utilize ubiquitous urban sensing data do not adopt a database approach which is arguably more convenient from a user perspective.

Our research on developing U^2SOD-DB is motivated by the following two aspects. First, we observe that Origin-Destination (OD) data is much simpler than the data types that are modeled by both classic GIS, SDB and MOD systems. An OD record basically has a pair of spatially and temporally referenced points besides additional associated attributes. An OD record can be modeled as a fixed sized relational tuple although the spatial and temporal components need to be handled specially for efficiency purposes. In contrast, both classic SDB and MOD have variable sized records to handle polygon/polyline/trajectories. Many bounding box based indexing techniques that are effective for polygons/polylines/trajectories become ineffective for OD data as the data volumes of bounding boxes can be as large as the data volumes of the OD data itself. Second, recent research works on in-memory based columnar store of relational data have shown that various in-database data compression and relational operations on multicore processors using parallel threads can achieve significant speedups [18, 19]. As many spatial and temporal operations on urban sensing data are both computational and I/O intensive, it is desirable to exploit the parallel processing powers of commodity hardware, including both multi-core CPUs and many-core GPUs.

It is easy to observe that GPS-recorded OD data is a subset of GPS trajectory data with only the first and the last GPS reading in a trip. Obviously adequately sampled GPS traces provide more spatiotemporal information than an OD pair. A GPS trajectory can actually be considered as the concatenation of consecutive OD pairs. However, we argue that OD data may have richer semantics than GPS traces as the origins and destinations of trips are explicitly marked to delineate the starting position/time and the ending position/time. Additional trip-specific attributes might have been attached to OD pairs when they are collected. In contrast, GPS traces are often needed to be segmented or clustered to identify trips based on various heuristics [20] or processed manually using a travel survey approach [21]. Furthermore, the data volumes of OD pairs are usually much smaller than those of GPS traces which make the OD data more suitable for large-scale studies. Although the ideal situation is to integrate the trip data with GPS traces in many applications, we argue that techniques that are designed for management and analysis of OD data can NOT be subsumed by those that are designed for GPS data. For cell phone call log data and location dependent social network data where the exact geometric traces are either not available or not important, techniques that are designed for OD data can be more suitable for such applications.

To the best of our knowledge, the work reported in this paper is the first in designing and implementing a high-performance data management system for large-scale ubiquitous urban sensing data. While it is difficult to develop a full-fledged system to handle all types of U^2SOD data, we have taken an incremental approach to developing modules for such purpose. Our current research and development are driven by the practical needs of processing large-scale taxi-trip records in NYC. We expect the accumulated modules will eventually lead to an integrated database system to lower the barrier of managing and data mining of large-scale U^2SOD data, so that

domain users can focus on urban research issues rather than computing techniques. This is arguably a significant contribution from the Urban Computing research community.

3 SYSTEM ARCHITECTURE AND IMPLEMENTATION DETAILS

Our design of U^{2}SOD-DB has three layers. The bottom tier is closely related to physical data layout and we have adopted a time-segmented, column-oriented data layout approach. The raw data are first transformed into binary representations and attributes are clustered into groups based on application semantics. The data corresponding to the attribute groups at a certain time granularity are stored as a single database file with the relevant metadata registered with the database system. We assume one or more database files can be streamed into CPU main memory as a whole to maximize disk I/O utilization. Multicore CPU processors can access the data files in parallel once they are loaded into the CPU main memory. They can also be transferred to GPU global memories should the system determine that GPU parallel processing is more advantageous. The middle tier is designed for efficient data accesses and aggregations, including compression, aggregation and indexing. While we skip the implementation details of most of the modules in this layer due to space limit, we would like to note that a few in-memory based indexing techniques for both categorical (e.g., trip mode and trip purpose), 1D numeric (e.g., time/distance/fare/tip) and 2D numeric data (mostly x and y coordinates of pickup/drop-off locations) are being explored. The third tier handles efficient joins between OD data and urban infrastructure data, such as road networks and administrative regions. The overall architecture is shown in Fig. 1. We next provide implementation details of several key components in the design.

3.1 TIME-SEGMENTED COLUMN-ORIENTED DATA LAYOUT

As shown in Fig. 2, we group the attributes that are associated with trip records into several groups and data of the attributes in the same group are stored in a single database file by following the column-oriented layout design principle - attributes in the same group are likely to be used together and thus it is beneficial to load them into main memory as a whole to reduce I/O overheads [16]. Given a fixed amount of main memory, as the attribute field lengths of individual groups are much smaller than the length of all attributes, more records that are related to analysis can be loaded into main memory for fast data accesses. In general, the column-oriented data layout design improves traditional tuple based physical storage in relational databases by avoiding reading unneeded attribute values into main memory buffers and subsequently increasing the number of tuples that can be read into the buffers in a single I/O request. We note that while traditional relational databases use page as the basic unit for I/Os (e.g., 8 kilobytes), modern hard drives often have large hardware caches (e.g., 32 megabytes). As such, reading large chunks of tuples can be advantageous for read-only data to fully utilize hardware capacity and improve overall disk I/O performance which is arguably the most severe bottleneck in processing large-scale data. We also note that combining several attribute groups into one and extracting attributes from multiple groups to form materialized views can be beneficial for certain tasks. For example, in Fig. 2, attribute group 5 (start_x, start_y, end_x and end_y) can be considered as a materialized view of the attribute group 2 (start_lon,start_lat, end_lon and end_lat) by applying a local map projection to the lat/lon pairs. Since the projected data are frequently used in calculating geometric and shortest path distances and map projections are fairly expensive, materializing attribute group 5 can significantly improve system performance. Another example to demonstrate the utilization of materialized views on the physically grouped attributes is verifying the recorded trip times (indicated by trip_time in group 6) with computed trip times (by subtracting pickup time from drop-off times in group 3) by materializing the respective attributes in the two groups. We further note that among the attributes in the original dataset shown in Fig. 2, some of them can be derived from others. For example, both start/end zip codes (group 9) and addresses of pickup and drop-off locations (group 8) can be derived from start/end latitudes and longitudes (group 2) through reverse geocoding or other related techniques.

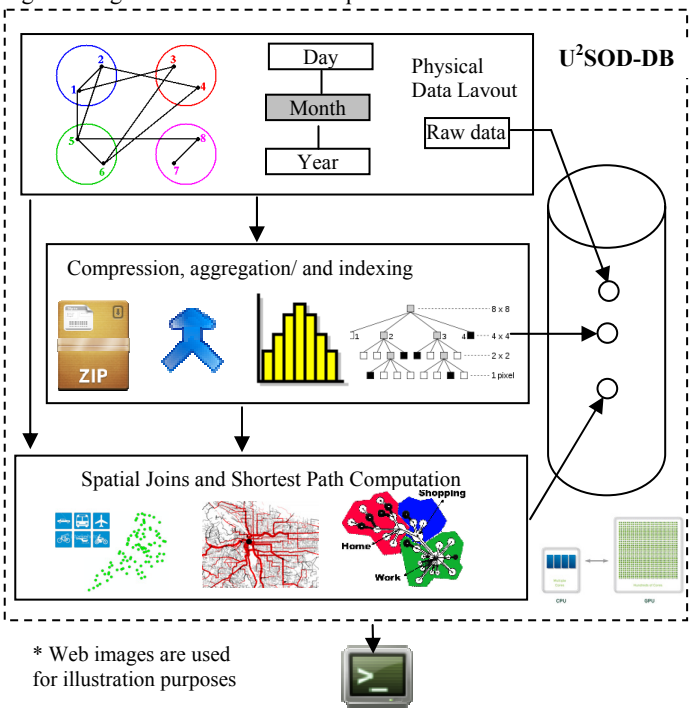


Fig. 1 U^{2}SOD-DB System Architecture

After determining data layout by grouping attributes (vertical partitioning), the next issue is to determine the appropriate number of records in database files so that these files can be efficiently streamed among hard drives, CPU main memory and GPU device memory. The sizes of the database files should be larger than those of hard disk caches but small enough to accommodate multiple database files simultaneously in CPU/GPU memories so that typical operations can be completed in the designated memory buffers. At the same time, the numbers of records should correspond to certain time granularities as much as possible. In our design, we use month as the basic unit (temporal granularity) to segment taxi trip records (horizontal partition). Given that there are about half million taxi trip records per day in NYC, assuming that an attribute group has a record length of 16 bytes (e.g., four attributes with each represented by a 4-bytes integer), the database file would be 16*0.5*30=240 megabytes. Since the hard drive cache and the main memory capacity in our experiment system are 16 megabytes and 16 gigabytes,

respectively, the database file size seems to be appropriate although other sizes might be suitable as well. We plan to develop a data layout advisor when U^{2}SOD-DB is applied to other types of OD data or applied in a different region by incorporating user-defined parameters.

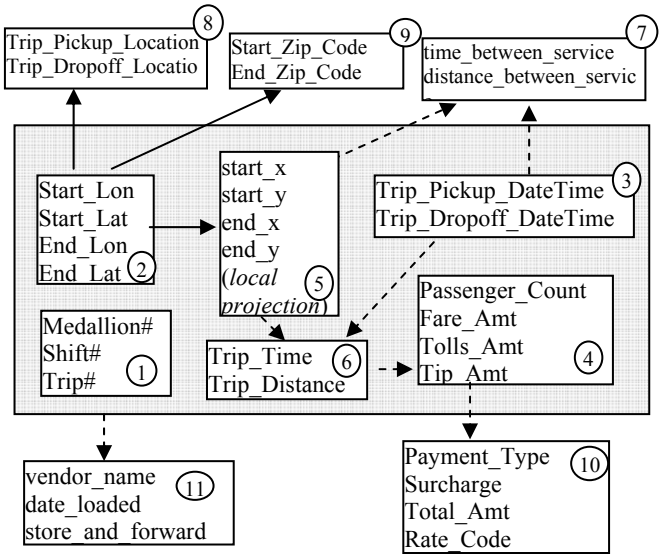


Fig. 2 Column-Oriented Physical Data Layout in U^{2}SOD-DB

3.2 Efficient Spatial -Temporal Aggregations

While many operations and analytical tasks can be performed on the U^{2}SOD data, spatial and temporal aggregations are among the fundamental ones. Fig. 3 illustrates the general framework and example spatial and temporal aggregations that are supported by our U^{2}SOD-DB design. Most of the temporal aggregations (e.g., daily and hourly) and some of the spatial aggregations (e.g., grid based) can use universal schemas, while more complex aggregations rely on the schemas provided by infrastructure data such as road network and administrative hierarchies. While we defer the discussions on associating OD data with the infrastructure data through spatial joins in the next subsection, we note that, as shown in Fig. 3, once the OD locations (i.e., pickup and drop locations in the taxi trip data) are associated with street segments or different types of zones, the aggregations can be reduced to simple grouping without involving expensive spatial and/or temporal operations any more. These relatively simple aggregations should be supported efficiently as much as possible to allow interactive visual explorations.

Towards this end, we have designed and implemented a GPU based spatial and temporal aggregation module in our U^{2}SOD-DB. As shown in the experiment section (Section 4), the GPU based aggregation module can significantly improve aggregation performance by 1-2 orders by making use of the massively data parallel computing power on GPUs [8] that are already widely available on commodity computers. The design uses four parallel primitives, namely *Transform*, *Sort*, *Reduce_By_Key* and *Scatter*. Take the grid-based spatial aggregation for an example, the *Transform* primitive converts x/y coordinate pairs into row-major ordered array indices and uses the indices as keys for the subsequent three primitives. The *Sort* primitive sorts the keys. As a single commodity GPU device can sort integers at the order of a few hundreds of millions to over a billion integers per second, this step can be efficiently processed for reasonably sized OD datasets in the orders of a few hundreds of millions on current commodity GPUs. The *Reduce_By_Key* primitive sums up the numbers of the same keys in the input vector and output the unique keys and their numbers of occurrences. In the grid-based spatial aggregation case, the numbers of pickup or drop-off locations within all the non-empty grid cells are computed. The last step essentially transforms the aggregation results from a sparse array representation into a dense array presentation should the dense array representation be more desirable. The step makes use of the *scatter* primitive and is optional. In the grid-based spatial aggregation case, a 2D grid is created and the numbers of non-empty grid cells are written to the corresponding positions by using the keys as the indices in the 2D grid (row-major ordered) while leaving the empty grid cells to have the initial zero value.

We note that the similar process can be applied to many aggregations in Fig. 3 as long as a key can be identified for aggregation. In addition to simply counting the numbers of records under a same key using the *Reduce_By_Key* primitive, many other types of operations can be implemented by using user defined binary functions, i.e., functors [22]. Given that GPUs are designed to support large-scale floating point computation, we believe it is promising to design more insightful but more computationally intensive measurements on GPUs during the aggregations. This is left for future work.

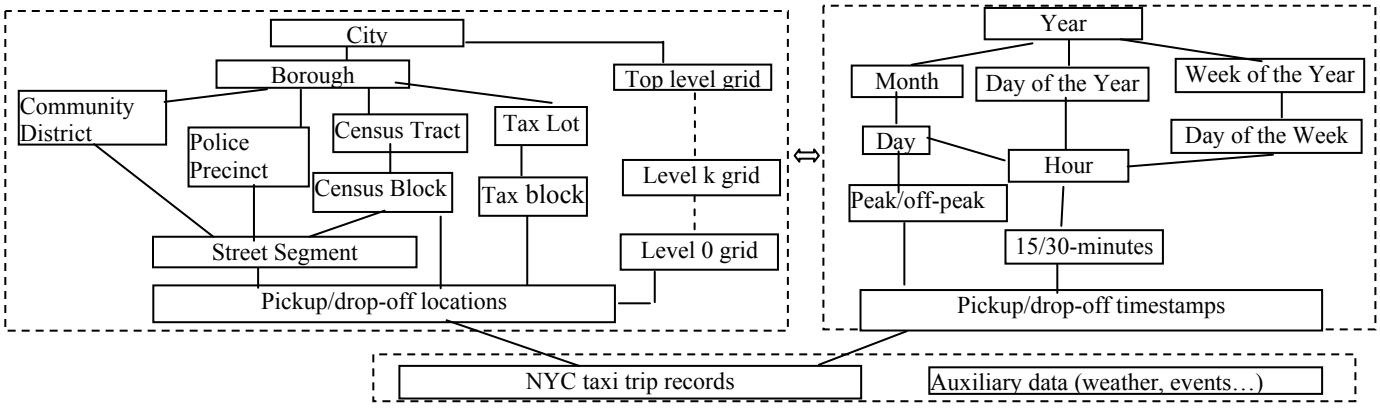


Fig. 3 Example Spatial and Temporal Aggregations in U^{2}SOD-DB

3.3 Spatial Join with Infrastructural Data

Very often OD data needs to be associated with infrastructure data, such as street networks, administrative regions and census zones. In NYC, the MapPluto tax lot dataset [23] has detailed land use information of each tax block. As tax blocks are typically at the finer spatial granularity than census blocks, they can be very useful in identifying trip purposes. In addition, in big cities such as NYC, usually there are large numbers of categorized Points of Interests (POI) sites that are collected by companies such as NAVTEQ for navigation purposes. While the infrastructure data are dynamic themselves, the change rates are relatively slow and can be considered static for reasonable long study periods. Associating OD data with these infrastructure data can be done offline to speed up online query processing in many cases. Based on the observation that inefficient disk I/Os dominate existing SDB and GIS systems for such spatial joins, we leverage the in-memory friendly physical data layout and our existing work on GPU-based indexing of large-scale point data [24] for this purpose. An open source spatial indexing package called libspatialindex [25] is integrated to develop a CPU-GPU hybrid approach to achieve the desired level of performance of spatial joins in U^{2}SOD-DB. Before introducing the design and implementation details, we would like to provide more details on the three types of spatial joins that U^{2}SOD-DB currently supports based on application needs, i.e., P2N-D, P2P-T and P2P-D.

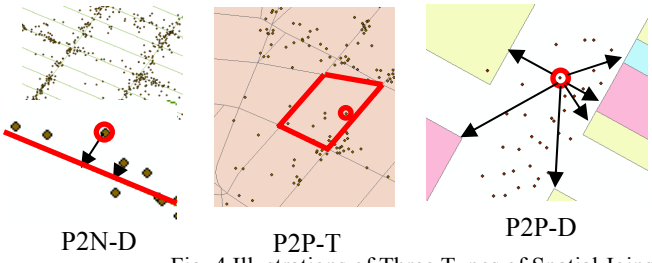


Fig. 4 Illustrations of Three Types of Spatial Joins between OD Data and Urban Infrastructure Data

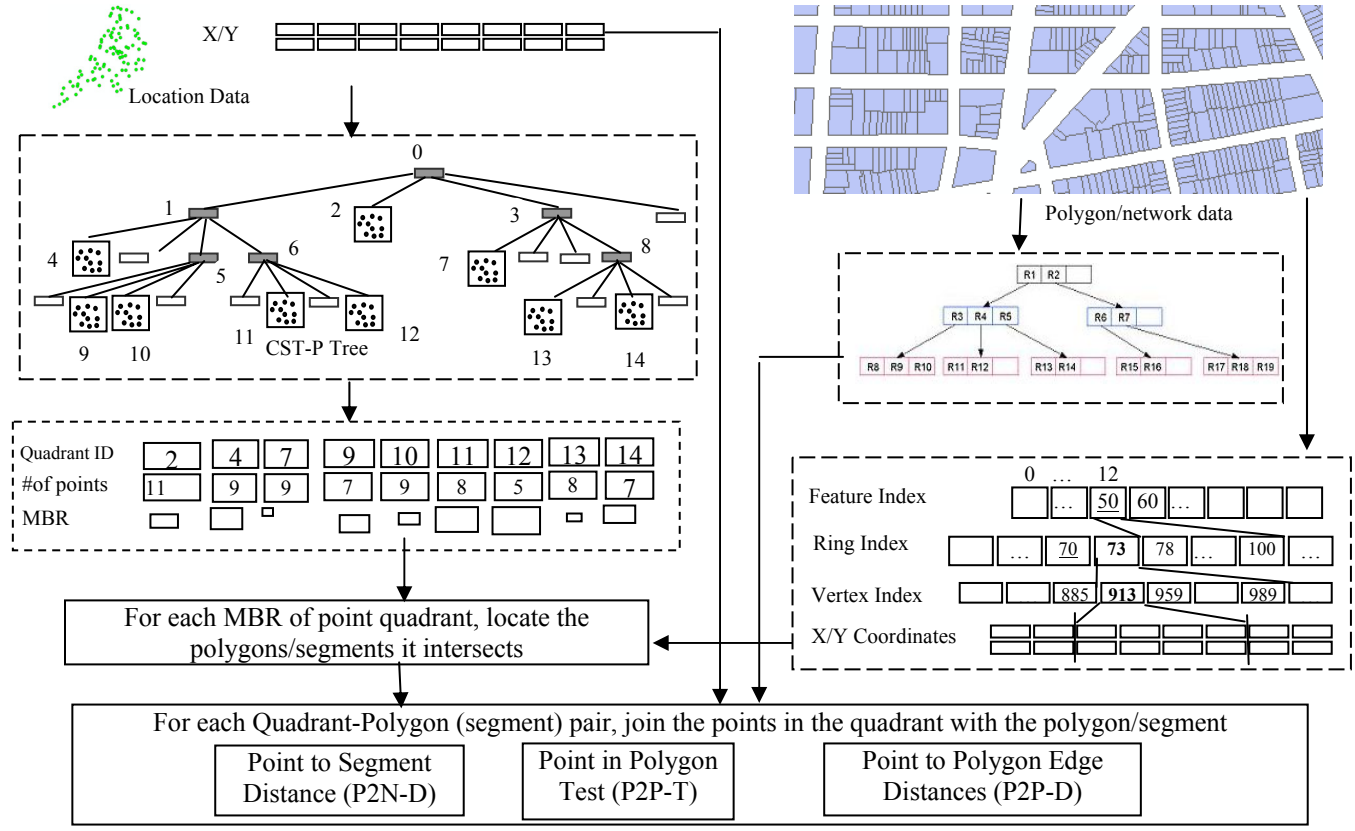


Fig. 5 Illustrations of Spatial Join Processing using a Hybrid Framework and Cache-Friendly In-Memory Data Representations

The P2N-D spatial join associates each point with its nearest street network segment. The P2P-T spatial join associates each point with a polygon that it falls within, i.e., through point-in-polygon test. The P2P-D spatial join associates a point with a set of polygons ordered by the smallest distance to the edges of such polygons. The three types of spatial joins are illustrated in Fig. 4. The purposes of P2N-D and P2P-T are straightforward and we next briefly explain the utilization of the P2P-D spatial join in U^{2}SOD-DB for processing taxi trip records. Conceptually, most of taxi pickup drop-off locations (or origins and destinations in general) should be close to street segments but outside of buildings and other types of properties that are not accessible to public vehicles. The distances between the locations and the boundaries of the properties can be important in helping identify trip purposes. However, the same O/D location can be associated with multiple polygonal property

boundaries and there are some inherent uncertainties of the approach. Since we are interested in aggregated trip patterns rather than individual trips, the uncertainties are not a problem in computing aggregated trip patterns. Compared with P2N-D and P2P-T spatial joins, P2P-D can also be useful in handling positional errors that are inherent to GPS or GPS-equipped devices, especially in NYC where urban canyon effect can be significant [21]. We note that P2P-D is related to KNN queries in classic SDB when the distance between a point and a polygon is used to define nearest neighbors. However, in addition to providing parameter K, P2P-D spatial join also has an influence window and only the top-K polygons that intersect with the influence window are joined with a query point. As such, P2P-D spatial join can be considered as a combination of window query and KNN query for each query point. From a computing perspective, P2P-D is more complex than P2N-D and P2P-T. This is because a set of (identifier, distance) pairs, instead of a single identifier, needs to be returned from this type of spatial join for each location in an OD record.

The proposed solution is a hybrid CPU-GPU approach. First, a Constrained Spatial Partition Tree for Point Data (CSPT-P Tree) is constructed on GPU using the Thrust parallel library [22] which has achieved a 23X speedup than a serial CPU implementation as detailed in our technical report [24]. Second, an R-Tree is constructed using the libspatialindex package [25] on CPU on the infrastructure data (street network segments or zones). Third, the bounding boxes of the set of points that fall into the leaf nodes of the CSPT-P tree are queried against the R-Tree to associate the leaf nodes with polygons. Assuming the influence window size is (w, h) and the bounding box is (x1,y1, x2, y2), then if the query window (x1-w,y1-h,x2+w,y2+h) intersects with the Minimum Bounding Rectangle (MBR) of a polygon/segment indexed by the R-Tree, then the CSPT-P tree node and the polygon are paired and queued for further processing. This is essentially the filtering phase in classic spatial joins [14]. The final step is to actually join the points with the polygon/segment. As a polygon/segment is a collection of points, this step essentially requires pair-wise computation among the points in the two datasets before applying an aggregation function g(x) to each of the query points. For P2N-D and P2P-D, g(x) is the minimum function on distance. For P2P-T, the function g(x) can be a boolean function on the number of intersections if a ray-casting algorithm is applied. As there are multiple polygons/segments returned from a query on the R-Tree, a second aggregation function f(x) is needed for reduction. For P2N-D, f(x) is the minimum function on distance. For P2P-T, f(x) is a boolean testing function as a point can only fall within a single polygon. For P2P-D, f(x) is the top-K ranking function on distances.

Among the differences between our spatial join framework and the existing SDB implementations (e.g., PostgreSQL), the most signficant one is that our framework exploits the cache-conscious array representations extensively for both OD location data and the infrastructure data as well as their auxiliary indices whereas possible. While existing SDB implementations utilize sophisticated data structures to efficiently map between database files and main memory buffers to reduce I/Os and minimize computation when the available main memory buffer is small, this is unnecessary in our framework as we aim at making full use of large memory capacities available to modern computing systems. Our time-segmented column-oriented physical layout and the data compression modules ensure that the memory footprint within a batch computing does not exceed available memory capacity. Compared with using memory pointers which is required in many dynamic data structures, the array based representation not only is cache friendly but also reduces memory footprints significantly. The technique is especially useful on modern hardware as data accesses are becoming much more expensive than computing [17].

As detailed in the evaluation section (Section 3), our implementation is able to complete a P2P-D spatial join of 150 million taxi pickup locations with 43 thousands MapPluto tax lot polygons in about 500 seconds using an influence window size of (100,100) feet. Analyzing the runtimes of the P2P-D join reveals that building the CSPT-P tree for the location data took only about 6-7 seconds on GPUS while nearly an order more time was spent on querying the intersecting polygons of CSPT-P tree quadrants and two orders more time was spent on distance computation. This clearly indicates that a more efficient spatial indexing approach for infrastructure data is needed in an in-memory computing environment. We are considering replacing the libspatialindex with an in-house GPU-based R-Tree implementation or using a simpler multi-level grid file approach. It is also relatively straightforward to parallelize distance computation on GPUs by assigning a quadrant-polygon pair to a GPU computing block and launching multiple threads in a computing block proportional to the number of points in a quadrant of the location data. Our preliminary results have shown more than 150X speedup on distance computation using GPU acceleration and we expect 10-40 times speedups for a pure GPU implementation. This will bring a total of 3- 4 orders of speedup (~2000X) when compared with the baseline serial CPU implementations using the state-of-the-art open source spatial indexing/query packages.

4 CASE STUDIES AND PERFORMANCE EVALUATIONS

4.1 Data and Experimental Setting

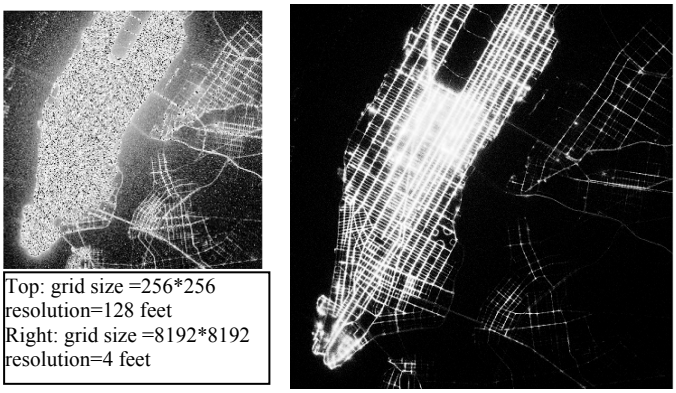
In addition to the taxi trip data with 300 million records in about two years that have been discussed above, our case studies also use several infrastructure datasets, including NYC DCPLION street network data and census 2000/2010 data that are publically available on the NYC-DCP website [23], and the NYC MapPluto Tax Lot data [23] which requires a license. Both the taxi trip data and the infrastructure data require several preprocessing steps before they can be used in the U^{2}SOD-DB system. The details of preprocessing are omitted due to space limit. All experiments are performed on a Dell Precision T5400 workstation equipped with dual quadcore CPUs running at 2.26 GHZ with16 GB memory, a 500G hard drive and an Nvidia Quadra 6000 GPU device. The sustainable disk I/O speed is about 100 megabytes per second while the theoretical data transfer speed between the CPU and the GPU is 4 gigabytes per second through a PCI-E card.

The taxi trip data are partitioned both vertically and horizontally as described in Section 3.1. Among the 11 attribute groups, only groups 3, 4, 5, 6 are used in this study. While more sophisticated analysis is possible, for example, analyzing profitability at the shift level, in this paper, we focus on two aspects. The first one is the essential functionality that is important to taxi trip analysis. In this case, the importance of performance is secondary to functionality and U^{2}SOD-DB will

be evaluated by its unique functionality. The second aspect is the performance of key operations, including both online aggregations and offline spatial joins. In this case, we will compare the performance of the U^{2}SOD-DB modules with the best-effort serial CPU implementations.

4.2 Performance Evaluations on parallel spatial and temporal aggregations

To evaluate the GPU-based spatial and temporal aggregation approach presented in Section 3.2, we feed the 177 million taxi trips in 2009 to the system with a spatial coverage of approximately 3*3 miles that covers the middle- and low-Manhattan at a 0.5 feet spatial resolution which results in over 150 million pickup locations in the area. While the GPU memory footprint of the algorithm depends only on the number of locations prior to generating a final grid with the value in each cell representing the number of pickup locations that fall within the cell, the GPU memory limit (6GB on Nvidia Quadro 6000 device) prevents us from experimenting CSPT-P tree levels 14 or above where the number of grid cells is 256 million (2^{28}) or higher. As such, we have performed grid-based spatial aggregations at the levels 8-13 (6 levels) using the 2D sparse matrix presentation with additional aggregations on levels 14-16 using a 1D vector representation (c.f. Section 3. 2). The resulting 2D grids at the levels 8 and 13 are shown in Fig. 6. The picture on the right clearly shows the effects of the street network on the taxi pickup locations.



Top: grid size =256*256 resolution=128 feet
Right: grid size =8192*8192 resolution=4 feet

Fig. 7 Grid-Based Spatial Aggregations of Pickup Locations in 2009 with two Resolutions

Since the grid-based spatial aggregations can also be easily implemented on CPUs by sequentially looping through all locations and increasing the counters at the corresponding grid cells, our evaluations focus on the performance gains due to GPU accelerations in U^{2}SOD-DB. First, the total GPU runtimes vary from 240 ms at level 8 to 326 ms at level 13, a 38.5% increment. In contrast, the CPU times increase from 2497 ms to 17786 ms, a 712.3% increment. The results clearly demonstrate the scalability of our GPU-based implementation. While GPU memory capacity currently is a limiting factor, we can coordinate the radix sort algorithm on both CPUs and GPUs to allow larger numbers of data points to be sorted which is left for future work. Second, more importantly, when comparing GPU total runtimes with CPU runtimes, we see a gain of 10-30X speedup as the grid levels increase. We would also like to note that our GPU implementation uses the Thrust parallel primitives [22]. Although very often a good tradeoff between coding complexity and code efficiency, the primitives based implementations usually are not the most efficient implementations achievable. By re-implementing the module using a native programming language to consolidate the four steps (c.f. Section 3.2) and use GPU shared memory efficiently, it is quite possible to further reduce GPU runtimes at all levels for higher speedups. We note that the CPU implementation only involves the transform and scatter steps without needing the sort and reduce steps which is only possible for sequential algorithms. In addition, the CPU implementation also has been optimized with –O3 compilation flag and thus we do not expect further performance improvements. We have also experimented temporal aggregations at the hour and minute temporal granularities using the same set of data after spatial filtering. The runtimes for the GPU and CPU implementations are 197.63 ms and 1709.02 ms, respectively, at the minute granularity with a bin number of 24*60=1440, i.e., a speedup of 8.6X has been achieved. The runtimes are slightly lower at the hour granularity (bin number = 24) where the runtimes are 165.12ms for GPU and 1598.89 ms for CPU, respectively. Again, a 9.7X speedup has been achieved.

While it seems that the runtimes for both spatial and temporal aggregations on CPUs are already acceptable in many offline applications, we argue that GPU accelerations can further make real time and interactive spatial and temporal aggregations possible by providing sub-second response times. As the experiments have shown, this currently is not possible on CPUs, especially in the case of high-resolution spatial aggregations. While using indexing and/or materialization can potentially also reduce the runtimes of CPU-based applications to sub-second level, our primitives based GPU implementation is preferable due to its simplicity. The implementation essentially requires only 4 lines of code with each line corresponding to a parallel primitive and a few lines to implement a functor (C++ functional object) to map between values and bin numbers (either a grid cell in spatial aggregations or a time index in temporal aggregations). When the aggregation rules get more complex, such as combinations of spatial and temporal queries, we expect our GPU based implementation will achieve higher speedups over CPU based ones. We leave the performance studies of complex spatial, temporal and spatiotemporal aggregations for future work.

4.3 Performance Evaluations on P2P-D Spatial Join

To test the efficiency of the P2P-D spatial join implementation, we have built a CSPT-P tree for the taxi trips whose pickup locations fall within the study area shown in Fig. 7. We limit the spatial extent of the study area mostly because our current CSPT-P tree construction algorithm can only calculate Morton codes from points with 32 bits (the extension is left for future work). We could have covered all NYC areas using a coarser spatial resolution. Since more than 85% of pickup locations of all the taxi trips in 2009 fall within the study area, we believe it is acceptable to limit our study area but use a higher spatial resolution (0.5 feet). For the CSPT-P tree construction, the only parameter to set is the maximum number of points that is allowed in the leaf nodes of the CSPT-P tree (N). Table 1 lists the end-to-end runtimes using three N values of1000, 2000 and 5000. We have used the default parameters for the R-Tree. The number of nearest neighbors is empirically set to 10. From the results listed in Table 1 we can see that our

current implementation is able to achieve an end-to-end runtime in the order of 500-2000 seconds using a variety of parameter combinations for P2P-D spatial join between the 150,417,865 pickup locations and 43,252 polygons. Compared with pure CPU implementation which requires about 30.5 hours to complete the same P2P-D join, we have achieved more than two orders (100X) of speedup. Given the involved computational intensities in terms of numbers of locations (A), quadrant-polygon pairs (D), point to polygon edge distance computations (E) and the resulting records (F) listed in Table 1, it is clear that modern processors are quite capable of processing large-scale U^{2}SOD data and there are great potential for parallel processing for larger-scale data.

Table 1 Computation Intensities and Runtimes of P2P-D spatial join on 150 million locations and 43 thousands polygons

Max # of points in a CSP-Tree quadrant		1000	2000	5000
	A	15,0417,865		
	B	893,871	458,737	166,601
	C (sec.)	7.422	7.184	6.829
Query Window Size w=100 h=100	D (million)	7.036	3.430	1.370
	E (billion)	15.401	12.309	13.623
	F (billion)	1.025	0.997	1.022
	G (sec.)	115.377	53.501	20.253
	H (sec.)	595.061	462.516	519.809
	I (sec.)	36.723	34.813	46.260
	K (sec.)	754.583	558.014	593.151
Query Window Size w=200 h=200	D (million)	20.015	9.850	3.752
	E (million)	29.312	25.848	27.078
	F (billion)	1.377	1,290	1.313
	G (sec.)	171.317	80.353	30.508
	H (sec.)	1,197.532	1,090.533	1,118.527
	I (sec.)	269.790	210.558	227.604
	K (sec.)	1646.061	1388.628	1383.468

(A) - #of pickup locations, (B) - #of CSPT-P Tree quadrants, (C) - CSPT-P Tree construction time (sec.) (D) - #of quadrant-polygon pairs (E)- #of point to polygon edge distance computation (F) - #of resulting records (G) - index query time (sec.) (H) data query time (sec.) (I) - result gathering time (sec.) (K)- total time (sec.) = C+G+H+I

7 CONCLUSION AND FUTURE WORK

In this paper, we reported our design and implementation of U2SOD-DB, a column-oriented, GPU-accelerated, in-memory data management system targeted at large-scale ubiquitous urban sensing origin-destination data. Experiment results show that U^{2}SOD-DB is capable of handling hundreds of millions of taxi-trip records with GPS recorded pickup and drop-off locations and times efficiently.

The accomplished work so far is preliminary in nature when compared to our ultimate goals. First, while we target at the general U^{2}SOD data when we abstract data models, design operations and develop efficient implementations, they may be specific to taxi trip data and thus more generalizations are needed. Second, while a few GPGPU based algorithms have been proposed for efficient implementations of essential functionality of the system, there are quite a few that still rely on existing packages that are serial CPU code. More research efforts are needed to make the system achieve even better performance by fully exploiting GPGPU technologies. Finally, a SQL front end is needed to make the system a true database system. These are left for future work.

REFERENCES

1. Reades, J., Calabrese, F., et al. 2007. Cellular Census: Explorations in Urban Data Collection. Pervasive Computing, IEEE 6(3): 30-38.
2. Calabrese, F., Colonna, M. et al. 2010. Real-Time Urban Monitoring Using Cell Phones: A Case Study in Rome. IEEE Transactions on Intelligent Transportation Systems 12(1): 141-151.
3. Vasconcelos, M. A., Ricci, S., et al. 2012. Tips, Dones and Todos: Uncovering User Profiles in Foursquare. Proceedings of the fifth ACM International Conference on Web Search and Data Mining.
4. Calabrese, F., Kloeckl, K., et al. 2008. WikiCity: Real-time Location-sensitive Tools for the City. In Handbook of Research on Urban Informatics: The Practice and Promise of the Real-Time City (Foth, M. eds) 390-413. IGI Global.
5. Friedland, G., Choi, J., et al. 2011. Video2GPS: A Demo of Multimodal Location Estimation on Flickr Videos. Proceedings of the 19th ACM International Conference on Multimedia.
6. Düntgen, C., Behr, T., et al. 2009. BerlinMOD: A Benchmark for Moving Object Databases. The VLDB Journal 18(6): 1335-1368.
7. Sakr, M. and Güting, R. 2011. Spatiotemporal Pattern Queries. GeoInformatica 15(3): 497-540.
8. Kirk, D. B. and Hwu, W.-M. W. 2010. Programming Massively Parallel Processors: A Hands-on Approach Morgan Kaufmann.
9. Zheng, Y., Liu, Y., et al. 2011. Urban Computing with Taxicabs. Proceedings of ACM UbiComp.
10. Phithakkitnukoon, S., Horanont, T., et al. 2010. Activity-Aware Map: Identifying Human Daily Activity Pattern Using Mobile Phone Data. Proceedings of the First International Conference on Human Behavior Understanding.
11. Zheng, Y., Chen, Y., et al. 2009. GeoLife2.0: A Location-Based Social Networking Service. Proceedings of the 10th Tenth International Conference on Mobile Data Management.
12. Yuan, J., Zheng, Y., et al. 2011. Where to Find My Next Passenger? Proceedings of ACM UbiComp.
13. Xie, R., Ji, Y., et al. (2011). Mining Individual Mobility Patterns from Mobile Phone Data. Proceedings of the 2011 International Workshop on Trajectory Data Mining and Analysis.
14. Shekhar, S. and Chawla S. 2003. Spatial Databases: A Tour Prentice Hall.
15. Harizopoulos, S., Liang, V., et al. 2006. Performance Tradeoffs in Read-optimized Databases. Proceedings of VLDB.
16. Abadi, D. J., Madden, S. R., et al. 2008. Column-stores vs. row-stores: how different are they really? Proceedings of ACM SIGMOD.
17. Hennessy, J. L. and D. A. Patterson (2012). Computer Architecture: A Quantitative Approach. Morgan Kaufmann.
18. Holloway, A. L., Raman, V., et al. 2007. How to Barter Bits for Chronons: Compression and Bandwidth Tradeoffs for Database Scans. Proceedings of the ACM SIDMOD.
19. Bakkum, P. and Skadron K., 2010. Accelerating SQL Database Operations on a GPU with CUDA. Proceedings of GPGPUs.
20. Zheng, Y., Li, Q., et al. 2008. Understanding Mobility based on GPS Data. Proceedings of the ACM UbiComp.
21. Chen, C., Gong, H., et al. 2010. Evaluating the Feasibility of a Passive Travel Survey Collection in a Complex Urban Environment: Lessons learned from the New York City Case Study. Transportation Research Part A: Policy and Practice 44(10): 830-840.
22. Thrust Parallel library. http://code.google.com/p/thrust/.
23. http://www.nyc.gov/html/dcp/html/bytes/applbyte.shtml
24. Zhang, J. and Gruenwald, L. 2012. Spatial Indexing of Large-Scale Geo-Referenced Point Data on GPGPUs Using Parallel Primitives. Technical Report. Online available at http://www-cs.ccny.cuny.edu/~jzhang/papers/TR-CSTP.pdf
25. Libspatialindex. http://libspatialindex.github.com/
26. Wang, L., Zheng, Y., et al. 2009. A Flexible Spatio-Temporal Indexing Scheme for Large-Scale GPS Track Retrieval. Proceedings of MDM.