High-Performance Geospatial Analytics in HyPerSpace

Varun Pandey Andreas Kipf Dimitri Vorona
Tobias Mühlbauer Thomas Neumann Alfons Kemper

Technische Universität München
{pandey, kipf, vorona, muehlbau, neumann, kemper}@in.tum.de

ABSTRACT

In the past few years, massive amounts of location-based data has been captured. Numerous datasets containing user location information are readily available to the public. Analyzing such datasets can lead to fascinating insights into the mobility patterns and behaviors of users. Moreover, in recent times a number of geospatial data-driven companies like Uber, Lyft, and Foursquare have emerged. Real-time analysis of geospatial data is essential and enables an emerging class of applications. Database support for geospatial operations is turning into a necessity instead of a distinct feature provided by only a few databases. Even though a lot of database systems provide geospatial support nowadays, queries often do not consider the most current database state. Geospatial queries are inherently slow given the fact that some of these queries require a couple of geometric computations. Disk-based database systems that do support geospatial datatypes and queries, provide rich features and functions, but they fall behind when performance is considered: specifically if real-time analysis of the latest transactional state is a requirement. In this demonstration, we present *HyPerSpace*, an extension to the high-performance main-memory database system HyPer developed at the Technical University of Munich, capable of processing geospatial queries with sub-second latencies.

1. INTRODUCTION

There has been a rapid advancement in research areas such as machine learning and data mining, which can be attributed to the growth in the database industry and advances in data analysis research. This has resulted in a need for systems that can extract useful information and knowledge from data. Data scientists use various data mining tools on top of databases for this purpose. To achieve lower latencies and minimize transmission costs between the database and external tools, it is necessary to move computation closer to the data. The current trend in database research is to integrate these various analytical functionalities that are useful for knowledge discovery into the database kernel. The goal is to have a full-fledged general-purpose database that allows big data analysis along with conventional transaction processing.

At the same time, there has been an emergence of data-driven applications. Companies like Uber, Lyft, and Foursquare have a need to create real-time applications, including alerting systems, that consider the most current state of their data, enabling *real world awareness*. Some of these applications have been enabled by the advent of the Internet of Things and the massive amounts of geotagged sensor data it generates.

There are publicly available datasets that can help in geospatial exploration. The New York City (NYC) Taxi Rides [6] dataset is a good example, but is only a sample of what is captured by the aforementioned companies. The dataset contains approximately 1.1 billion taxi rides taken in the city since 2009. This represents about 470,000 taxi rides everyday in one of the most densely populated cities in the world. Uber, a popular on demand car service available via a mobile application, has also made a subset of the taxi rides available for the cities of San Francisco and NYC. For NYC, Uber published data containing around 19 million rides for the periods from April to September 2014 and from January to June 2015 [9]. Ever since the datasets were published, there have been multiple static analyses on these datasets [10, 2, 9]. The authors of [1] present a comprehensive system built from scratch for storing, querying, and visualizing geospatial data using kd-trees. Their system takes two seconds to execute a query that returns 100,000 taxi trips, which is too slow to address real-time workloads. MemSQL has some real-time capabilities [7] and is one of the first main-memory database systems (MMDBs) to deeply integrate geospatial support. The current database systems do not offer the performance required by real-time applications, and companies are often forced to build their own solutions [4]. We estimate that a 10x performance improvement is needed in general-purpose database systems to enable such applications/analytics.

We want to offer high-performance geospatial processing in a general-purpose database system that meets the requirements of real-time workloads, which can be used by emerging applications and data scientists alike without having to build their own system or use external tools for data analytics. Recent advancements in MMDBs research make it possible to efficiently create snapshots of the current database state. With our proposed system called *HyPerSpace*, we built a first prototype into that direction. Our goal is to drasti-

cally improve the performance of geospatial data processing in relational database systems by carefully using advanced encoding schemes and index structures. In our demo, we will present a web-based prototype called *HyPerMaps* that shows that it is possible to have an interactive analysis on geographical data using a general-purpose database system instead of a custom hand-written solution.

PostGIS [8] is a spatial database extension for the PostgreSQL object-relational database system. It adds support for geographic objects allowing users to formulate geospatial queries in SQL. PostGIS adds two popular spatial datatypes to PostgreSQL: *geometry* and *geography*. The *geometry* datatype treats the earth as a two dimensional flat surface. The earth is projected onto a plane and geographical coordinates are mapped to a two dimensional cartesian coordinate system. When evaluating spatial predicates such as *ST_Covers*, *ST_Intersects* and spatial measurements such as *ST_Area*, *ST_Distance*, this datatype allows for high efficiency, however, it comes with a drawback. Since it treats the earth as a two dimensional plane, the computations are not precise over a large area as the spherical nature of the earth is not considered. To put this into context, consider an example of the shortest distance between two points. In a 2D cartesian plane, the shortest distance between two points is a straight line, while on a spheroid the shortest distance between two points is a geodesic (shortest distance on the great circle). In contrast to *geometry*, the *geography* datatype treats the earth as a three dimensional spheroid and all the computations are based on the spheroid. The computations are precise since they are done on a spheroid, but they are very slow compared to those on *geometry*.

We implemented the *geography* datatype and the corresponding geospatial predicates, such as *ST_Covers*, in the high-performance hybrid OLTP and OLAP MMDB HyPer [3][1]. We achieve much better performance compared to an open-source database PostgreSQL, a commercially available MMDB (System A), and a successful key-value store (System B). This demonstration presents *HyPerSpace* and showcases that an interactive analysis of huge amounts of rapidly changing geospatial data is indeed possible.

2. HYPERSPACE

Similar to what PostGIS is to PostgreSQL, *HyPerSpace* is a geospatial extension to HyPer, a high-performance relational MMDB that is optimized for modern multi-core CPUs. HyPer belongs to an emerging class of hybrid databases, which enable *real world awareness* in real time by evaluating OLAP queries directly in the transactional database. In HyPer, OLAP is decoupled from mission-critical OLTP either by using the *copy on write* feature of the virtual memory management or *multi version concurrency control* [5]. These snapshotting mechanisms enable *HyPerSpace* to evaluate geospatial predicates on rapidly changing datasets.

For geospatial data processing in *HyPerSpace*, we make use of the Google S2 geometry library[2]. This is not novel, since System B also uses the S2 library for evaluating geospatial predicates. The novelty of our system is the integration of geospatial functionalities into a high-performance MMDB with snapshotting mechanisms which makes it possible to evaluate geospatial predicates on rapidly changing datasets.

At the moment, we support the three geospatial datatypes `Point`, `LineString`, and `Polygon`. Most of the geospatial processing is done using the S2 library.

S2 decomposes the earth into a hierarchy of cells. It considers earth of radius 1, and encloses it in a cube that completely covers it. S2 projects a point on the earth's surface onto one of the cube's faces and finds the cell that contains it. The faces of the cube are the top level cells, which can be recursively divided into four children to obtain lower level cells. There are 30 levels in total, and cells at the same level cover equivalent areas on earth (e.g., level 30 cells cover approximately $1cm^2$ each). The cells are enumerated using the Hilbert space-filling curve. The Hilbert curve is hierarchical in nature and fits well with the decomposition of earth into cells. Hilbert space-filling curves are fast to encode/decode and they have a very desirable spatial property: they preserve spatial locality. This means that the points on earth that are close to each other are also close on the Hilbert curve. The enumeration of the cells gives a compact representation of each cell in a 64 bit integer called *CellId*. A *CellId* thus uniquely identifies a cell in the cell decomposition. Similarly, other spatial datatypes like `LineString` and `Polygon` can be approximated using cells.

The enumeration of cells in S2 is hierarchical, which means that a parent cell shares its prefix with its children. To check if a cell is contained in another, we simply need to compare their prefixes, which is a bit operation. This enables one to index points based on their *CellIds* and thus be able to retrieve points contained in a certain cell by performing a prefix lookup on the index. B tree data structures are a good choice to index *CellIds*, since they support fast prefix lookups (essentially range scans). Additionally, B trees allow for high update rates, which is an essential requirement for real-time workloads.

3. EVALUATION

All experiments were run *single threaded* on an Ubuntu 15.04 machine with an Intel Xeon E5-2660 v2 CPU (2.20 GHz, 3.00 GHz maximum turbo boost) and 256 GB DDR3 RAM and all reported performance results are averages over ten runs.

For evaluation, we used the NYC Taxi Rides dataset consisting of approximately 1.1 billion rides taken in the city from January 2009 until June 2015. The dataset includes the pickup and dropoff locations (latitudes and longitudes), pickup and dropoff times, and various details about the trip, such as distance, payment type, number of passengers, various taxes, tolls, surcharge, tip amount, and total fare. For privacy reasons, it does not contain details about drivers or passengers. The exact route taken for the trip is also not available. We needed to clean the dataset as some of the pickup or dropoff locations did not make sense as they were way outside NYC. We cleaned such records from the dataset and only considered rides that originated between longitude values -70.00 and -80.00, and latitude values 35.00 and 45.00. For evaluation, we made use of the taxi data for the month of January 2015. The cleaned dataset for January 2015 contains a total of 12505344 records.

We compared *HyPerSpace* with the following related systems: System A, System B, and PostgreSQL 9.4.5 (postgis-2.2.0). Since PostgreSQL does not support intra-query par-

[1]When saying HyPer, we are referring to the research version of HyPer developed at the Technical University of Munich.
[2]https://code.google.com/archive/p/s2-geometry-library/

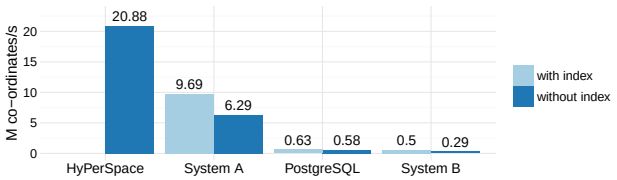

Figure 1: *HyPerSpace* vs. related systems: throughput of *ST_Covers* using lat/long co-ordinates

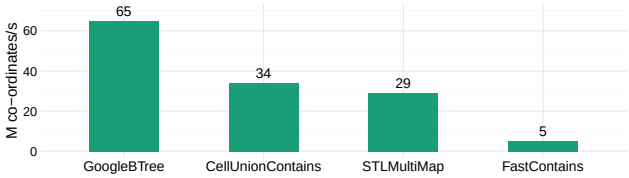

Figure 2: Microbenchmark results: throughput of *ST_Covers* using lat/long co-ordinates

allelism, we configured all systems to run single threaded. For evaluation purposes, we find how many rides originated from Midtown Manhattan in January 2015. In SQL notation, the following query is issued:

```
select count(*)
from   nyc,pickups_jan_2015
where  ST_Covers(nyc.geog,pickups_jan_2015.geog)
       and borough='Manhattan'
       and neighborhood='Midtown';
```

With the exception of System B, with NoSQL syntax, the query looks similar on all systems.

Figure 1 shows the throughput of the *ST_Covers* predicate for all of the systems. System A, System B, and PostgreSQL achieve better performance when using appropriate index structures. Particularly System B, which also makes use of the Google S2 geometry library, benefits from its index on points. System B's index is basically a B tree on the 64bit *CellIds*. System B computes an exterior covering of the polygon using the S2 library. That covering consists of cells at various levels (i.e., of different sizes). For each cell of this covering, it then performs a prefix lookup in the B tree (essentially a range scan) and evaluates qualifying points for actual containment in the polygon. System B suffers heavily from its document-based storage layout, since it needs to parse GeoJSON documents at runtime.

HyPerSpace completes the query in 550ms and thus achieves more than twice the performance of its closest competitor, which is System A with an index on points (1290ms). We have not evaluated *HyPerSpace* with an index on points yet, but ran multiple microbenchmarks outside of *HyPerSpace*. All microbenchmarks were implemented in C++11 and compiled with gcc 4.9.2 with `-O3` and `-march=native` settings. We compared the implementation *CellUnionContains* that we used in *HyPerSpace* as well as *FastContains*, which is a modified version of the `S2Loop.Contains` implementation that skips the initial bounding box check, to the two index-based implementations *GoogleBTree* and *STLMultiMap*.

Figure 2 shows the throughput of the *ST_Covers* predicate for the different implementations. *GoogleBTree*, which is an implementation similar to System B's index, completes the workload in 191ms. In the *GoogleBTree* implementation, we first compute exterior and interior coverings for the given polygon and then perform a range scan in a Google B tree[3] for each cell of the exterior covering. For each qualifying point, we check whether the point is contained in the interior covering, which is essentially a binary search on a sorted vector of *CellIds*. Only if a point qualifies the exterior, but not the interior covering, an exact containment check using our modified implementation of the `S2Loop.Contains` function needs to be performed. The other index-based implementation *STLMultiMap* takes twice as long (425ms) as *GoogleBTree* to complete the workload, even though it uses the same approach. In C++11, the `stl::multi_map` interface that we used in this case is implemented by a RB tree, which is less efficient for range scans. It is well known that a B+ tree would yield even higher rates for range scans than a B tree. However, for the sake of expediency and reproducibility of our measurements, we have used the B tree implementation provided by Google instead of a custom B+ tree implementation. Once we integrate this approach into *HyPerSpace*, we will make use of an optimized B+ tree implementation. The difference in performance between the two implementations *GoogleBTree* and *STLMultiMap* shows that the overall runtime of this approach is heavily influenced by the actual index structure used.

The approach *CellUnionContains* completes the workload in 367ms, compared to 550ms when implemented within *HyPerSpace*. The overhead is mostly caused by function calls that are issued for each of the 12M points. *CellUnionContains* is a straightforward approach. It first computes the bounding box and exterior and interior coverings for the given polygon. For each of the points, *CellUnionContains* then performs the following steps: First, it checks whether the point is within the bounding box. If that is the case, it checks for containment in one of the cells of the exterior covering. Analogous to the containment check for the interior covering, this essentially comes down to a binary search. Then the *CellUnionContains* approach continues analogous to the *GoogleBTree* approach by checking the interior covering and performing the exact containment check if necessary. By properly using the S2 mechanisms, our *CellUnionContains* approach achieves a slightly better performance than the index-based *STLMultiMap* approach, even though we have to loop over all of the 12M points.

4. DEMONSTRATION

We created an interactive web interface, called *HyPerMaps*, that demonstrates the outstanding geospatial processing performance of *HyPerSpace* on the NYC Taxi Rides dataset. The user interaction concept of *HyPerMaps* is designed to minimize the requirement of users' expertize with the explored data. The ability of *HyPerSpace* to answer queries with typically sub-second latency enables tight feedback loops. It supports users during query formulation and encourages an iterative approach. During filtering of the data, users can rely on datatype dependent elements, which provide context-based information like value distributions or geographic locations in real time. Users can draw polygons on the map to filter points geographically. Subsequently, users can combine different graphical and textual represen-

[3]https://code.google.com/archive/p/cpp-btree/

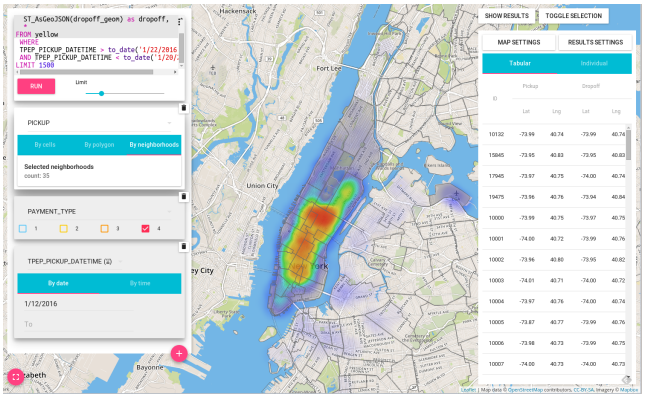

Figure 3: Interactive visualization of a real-time replay of NYC taxi rides using *HyPerMaps*

tations to create an informative and intuitive visualization. During this data exploration process, *HyPerMaps* will automatically compute updated results reflecting the current state of the user interface as well as the underlying dataset.

Figure 3 shows *HyPerMaps* visualizing the taxi dataset. On the left, various tiles allow users to specify filters on the data, which will be immediately translated into SQL code as illustrated on the top. This binding works in both directions—manually written SQL code will be translated into corresponding tiles. Users can choose between a heat map and pins to display selected points on the map. On the right, *HyPerMaps* shows aggregated information about selected points in tabular or in chart form.

During the demonstration we will run a *HyPerSpace* instance on a demonstration laptop with two months of data already loaded and one month of data being replayed at various speeds. We will motivate our demonstration by showing a scenario of a hotel owner wanting to install a shuttle bus for his or her guests to allow them to save money on their arrival, departure, and daily travels by not having to take expensive taxis. The hotel owner will select the taxi pickup area of his hotel by drawing a polygon on *HyPerMaps*, which boils down to a similar database query that we previously used for the evaluation of the different systems. Then, common dropoff areas of the taxi rides originating from this area will be displayed and the hotel owner can retrieve aggregate information for each of these areas. *HyPerMaps* will compute how much money and CO_2 emissions would be saved on aggregate if the hotel owner would install a shuttle to one of these dropoff areas. Additionally, we will demonstrate various scenarios showing that recently ingested data will be immediately reflected in *HyPerMaps*, thereby enabling *real world awareness* in real time.

5. TAKE-AWAY MESSAGE

In this paper, we presented *HyPerSpace*, a geospatial extension to the MMDB HyPer. Our implementation of the *ST_Covers* predicate achieves a much lower latency than corresponding implementations in related systems, without using any index structures. Additionally, we found that using index structures optimized for range scans such as B trees or B+ trees on *CellIds*, can yield even lower latencies. In our demo, we will show that it is indeed possible to build real-time visualizations on geographical data using a general-purpose database system instead of a custom hand-written solution that takes much longer to build and is harder to maintain. The novelty of our system is the integration of geospatial functionalities into a high-performance MMDB that allows for efficient snapshotting of the current database state. Our contribution also includes the careful use of the features of the Google S2 geometry library, thereby achieving much lower latencies than related systems. This makes it possible for the first time to evaluate inherently complex geospatial predicates on high throughput data streams in real time. To demonstrate this, we created a web interface that allows users to interactively explore the NYC Taxi Rides dataset while the data is being replayed at various speeds. Our demo shows that features (such as value distributions) of the entire dataset, including the most current data, can be used to populate UI elements, thereby supporting users in creating meaningful (aggregated) real-time visualizations.

6. ACKNOWLEDGEMENTS

This work is part of the TUM Living Lab Connected Mobility (TUM LLCM) project and has been funded by the Bayerisches Staatsministerium für Wirtschaft und Medien, Energie und Technologie (StMWi).

7. REFERENCES

[1] N. F. Ferreira, J. Poco, H. T. Vo, J. Freire, and C. T. Silva. Visual Exploration of Big Spatio-Temporal Urban Data: A Study of New York City Taxi Trips. In *IEEE Transactions on Visualization and Computer Graphics*, pages 2149–2158, December 2013.

[2] R. Fischer-Baum and C. Bialik. *Uber Is Taking Millions Of Manhattan Rides Away From Taxis.* http://fivethirtyeight.com/features/uber-is-taking-millions-of-manhattan-rides-away-from-taxis/.

[3] A. Kemper and T. Neumann. HyPer: A Hybrid OLTP & OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *ICDE*, pages 195–206, Apr. 2011.

[4] J. Miranda. *Uber Unveils its Realtime Market Platform.* http://www.infoq.com/news/2015/03/uber-realtime-market-platform/.

[5] T. Neumann, T. Mühlbauer, and A. Kemper. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *SIGMOD*, SIGMOD '15, pages 677–689, New York, NY, USA, 2015. ACM.

[6] *TLC Trip Record Data.* http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml.

[7] G. Orenstein. *Real-Time Geospatial Intelligence with Supercar.* http://blog.memsql.com/real-time-geospatial-intelligence-with-supercar/.

[8] *PostGIS.* http://postgis.net/.

[9] T. Schneider. *Analyzing 1.1 Billion NYC Taxi and Uber Trips, with a Vengeance.* http://toddwschneider.com/posts/analyzing-1-1-billion-nyc-taxi-and-uber-trips-with-a-vengeance/.

[10] C. Wong. *FOILing NYC's Taxi Trip Data.* http://chriswhong.com/open-data/foil_nyc_taxi/.