A Parallel Query Engine for Interactive Spatiotemporal Analysis

Mihir Sathe
Dept. of Computer Science
University of Southern California
msathe@usc.edu

Craig Knoblock
Information Sciences Institute
University of Southern California
knoblock@isi.edu

Yao-Yi Chiang
Spatial Sciences Institute
University of Southern California
yaoyic@usc.edu

Aaron Harris
Dept. of Mechanical Engineering
University of Southern California
arharris@usc.edu

ABSTRACT

Given the increasing popularity and availability of location tracking devices, large quantities of spatiotemporal data are available from many different sources. Quick interactive analysis of such data is important in order to understand the data, identify patterns, and eventually make a marketable product. Since the data do not necessarily follow the relational model and may require flexible processing possibly using advanced machine learning techniques, spatial databases or similar query tools do not make the best means for such analysis. Moreover, the high complexity of geometric operations makes the quick interactive analysis very difficult. In this paper, we present a highly flexible functional query engine that 1) works with multiple schema types, 2) provides low response times by spatiotemporal indexing and parallelization, 3) helps understand the data using visualizations, and 4) is highly extensible to easily add complex functionality. To demonstrate its usefulness, we use our tool to solve a real world problem of crime pattern analysis in Los Angeles County and compare the process with some other well known tools.

Categories and Subject Descriptors

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

General Terms

Design, Languages, Performance, Experimentation, Human Facors, Algorithms

Keywords

spatiotemporal analysis, spatial join, parallelization, indexing, visualization

1. INTRODUCTION

Finding spatiotemporal patterns is much like criminal investigation. You collect, integrate and aggregate the data, make some queries to find some leads. You further drill down to make some educated guesses about possible patterns and then you query further to test your guesses. Low response time queries are very important for such tasks. High complexity of most geometric operations increases the response time of such queries. Another important requirement for such analysis is quick visualization which ranges from mapplotting to visualizing non-spatial quantities like aggregations and frequency distributions. Moreover, this analysis often needs advanced operations like anomaly detection and clustering which are not strictky spatial or temporal.

Through this paper, we present a unique system that provides efficient query processing through in-memory indexing and parallelization. It also provides data visualization and can be extended easily to add advanced functionality. Our tool is deployed as a web application. Figure 1 shows the User Interface (UI) for our tool. UI is divided into three different panels: Top left panel to write queries, Right panel for visualizations and bottom panel for tabular and map view.

In the following section, we provide a motivating example of the analysis of statistical crime data in Los Angeles County. Section 3 contains details about the query model and capabilities of our system followed by description of the architecture in section 4. In section 5, we demonstrate the use of our tool to solve the problem described in section 2. We then evaluate our system in solving the said problem as compared to other well-known tools. Finally, we describe the related work and conclude with discussion of our contribution and future scope of the project.

2. MOTIVATING EXAMPLE

Strategic crime analysis is the study of crime information integrated with sociodemographic and spatial factors to determine long term "patterns" of activity [**book1**]. We ob-

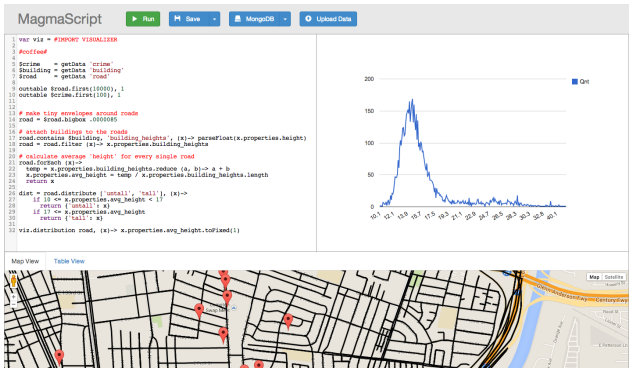

Figure 1: User Interface for our tool

tained data about a total 148,638 crimes in Los Angeles (LA) County since October 1st 2013 from the LA County Sheriff's Department [**lasd**] and various city level police departments [**mapping**]. Most crimes include latitude and longitude, date and time of occurrence, date and time of report and category of the crime (provided by individual agencies). We also have data about the geographical borders of cities in the county. We will analyze the crime patterns in the cities of Compton and West Hollywood. Figure 2 shows the heat maps of the crimes in those cities. Red spots indicate the highest concentration of the crimes.

Crimes being spatiotemporal, strategic crime analysis makes a great use case to demonstrate the usefulness of our tool. As a motivating example, we find the types of crimes that occur spatiotemporally together frequently. In other words, we seek the types of crime that frequently occur near each other within a short span of time. These crimes are more likely to be related to same entity (e.g. person, gang or a riot), this analysis helps us profile the behavior of such an entity and hence can give better insight to law enforcement agencies about how to deal with such an entity. For example, if we find that drunk driving and assaults occur together frequently in a certain city, strict enforcing of driving laws near the places that serve alcohol will help reduce the number of assaults as well.

The first part of this problem involves identifying the crimes occurring near each other within a short interval of time. Once we find all such series of crimes, the second part will involve analysis of the association rules between the categories of those crimes. We demonstrate solution of this problem using our tool and present the results in Section 5. In Section 6, we compare two different approaches of solving this problems: one involving use of our tool and the other involving the use of PostGIS and spatial extensions to the R language. We compare both approaches in terms of performance.

3. QUERY MODEL

We try to make queries easy to read and write by using higher order functions. These functions take stateless 'example' functions as parameters and apply them on entire dataset. This makes queries concise and parallelization ready. We allow users to write queries i JavaScript, CoffeeScript or a combination of both.

3.1 Spatiotemporal Operations

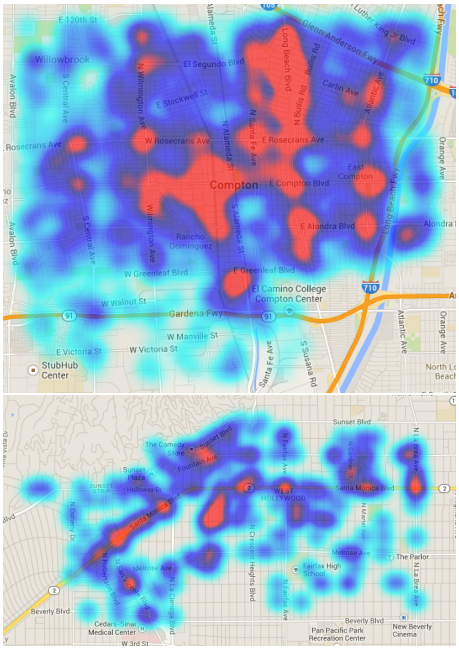

Figure 2: Crime heatmaps in the cities of Compton (above) and West Hollywood (below)

Figure 3: Use of contains operator in spatial joins

Our tool supports all common spatiotemporal operations like joins and aggregations. More complex and domain-specific functionality can be added to this tool as an extension. We explain spatiotemporal joins in greater details in this paper.

3.1.1 Spatial Joins

All the spatial join queries between two collections[1] can be written in the following format:

<collection 1> . <operator> (<collection 2>, <label>, <selector>)

Collection1 here is the collection on the left side of this join (note that all joins are left outer joins) and **operator** is the matching factor for the join (contains, contained by, intersects etc.). **Collection2** is the collection on the right side of the join. If an object from **Collection2** satisfies the criteria for join with an object in **Collection1**, part of the object of **Collection2** returned by the **selector** function is added to the said object of **Collection1** under the key specified by parameter **label**.

For example, if 'Compton' is one of the objects in the collection 'city' and crime with ID 1001 (a burglary) and crime with ID 1002 (a robbery) are objects in the collection 'crime' and both these crimes occurred inside Compton, Figure 5 shows how the spatial join with 'contains' operator will work.

We supported spatial operators like contains, covers and

[1]Throughout this paper, 'collection' refers to a data structure that is an ordered list of objects. This is loosely similar to idea of relational tables except collections are not relational. Also, different objects in collections can have different schema structure

```
//MATH.ext
(function() {
   return {
      add: function(a, b) { return a+b; },
      subtract: function(a, b) { return a-b; }
   }
})();

//Script
var math = #IMPORT MATH
outtext ( math.add(10, 15) ) // will print 25
```

Figure 4: Procedure to create a simple extension

intersects and temporal operators like together, within and around.

3.2 Extensions

Users can write their own extensions to any the desired functionality to the tool allowing them to use this tool as an integrated environment for analysis. Moreover any existing algorithm implementation in JavaScript or CoffeeScript can be imported as an extension with little to no modification. Figure 7 shows a simple MATH extension and the procedure to import in in your query. Some of the examples of the extensions made are function profiling, hierarchical clustering, spatial correlation, naive Bayesian classifier (see figure 6) and apriori association mining (used in section 5).

4. ARCHITECTURE

In this section we will describe the data architecture, execution environment and client-server interaction. We will also discuss the parallelization and parallel query infrastructure that we've built to run queries faster.

4.1 Data Format

All the data is handled in the GeoJSON[**geojson**] format internally. We have added some specifications to add temporal properties to GeoJSON objects. At the top level of object, we can add 'dtime' (along with 'properties', 'geometry' and 'type'). dtime contains 'start' and 'end' as time objects. If an object does not have an interval, it won't have the 'end' object. Figure 8 shows a spatiotemporal GeoJSON object that will work with our tool.

4.2 Data Architecture

Users can import their data from relational databases, CSV files, GeoJSON files or MongoDB collections. All data is converted to the said spatiotemporal GeoJSON format and stored in in-memory data store. Data can also be exported to same sources.

4.3 Execution Environment

Users write queries in a browser based code editor. When user runs a script, ther code is sent to a JavaScript sandbox on the server that contains in-memory data store and has full access to the query modeldescribed previously. Code from all imported extensions is also added to the sandbox. Asynchronous operations keep the clients waiting using KEEP ALIVE signal until the response is ready.

The response contains four types of output:

- **Text** to be displayed on the console

Figure 5: Architecture and request-response model

Figure 6: Workflow at the server side

- **Shapes** to be displayed on the map
- **Visualizations** to be displayed in the visualization window
- **Data** to be displayed on the output table

4.3.1 Parallel Processing Infrastructure

We have built a parallel infrastructure to run the queries in parallel. The main challenge here is to put all the pieces of the output together once the process is complete. The execution pipeline consists of a processing unit that fetches one instruction at a time from the execution queue, decides what collections to split and sends it to the processes to execute based on availability of data with that process (see subsection 4.3.2). Every query or job has a unique job identifier (ID) that helps identifying the result. Once a process is done executing its part of the job, it passes on the results to the result unit which keeps track of the number of expected process outputs and number of outputs recieved. Once all the outputs are recieved, it updates the in-memory store with the new output (if required) and notifies the processing unit of the completion by giving it the callback. Processing unit will look for next statement in the execution queue for the same job and proceed in the same way. If there are no more statements to be executed, it will end the processing and server will send the response with all the results upon next request by the browser. Figure 11 shows the server side workflow including the parallel processing.

4.3.2 Process Cacheing

Data can be passed to a new process by message passing over Transmission Control Protocol (TCP) connection which can carry large amount of JSON encoded data. However, considering the fact that 1] we are working on same collections again and again and 2] message passing for large data (few hundred MBs) is fairly time consuming, it does not make sense to send all the data every single time an operation is required over it. We therefore introduce an in-process cacheing of data. Every single process when spawned contains a Least Recently Used (LRU) cache that can cache the collections upon request. The cache also has a message passing interface for master thread to communicate with it. This helps our tool to make best use of available memory to give faster results.

4.4 Indexing

Indexing is the heart of efficient spatiotemporal operations. We provide existing in-memory implementations of R-Tree[**rtree**] for the spatial indexing and Interval-Tree[**clrs**] for temporal indexing of the objects. Indexes are created on all the spatial and temporal collections upon their creation unless otherwise specified. This is also true for the smaller collections that are created by splitting the larger collections for parallel processing. Since these indexes are in-memory, they do not take a lot of time to build on small-medium size data. Like the collections themselves, indexes

are also cached in the threads which helps reduce the overall execution time.

5. PROBLEM SOLVING

In this section, we demonstrate usefulness of our tool by using it to solve the problem proposed in section 2. As mentioned before, we have data about crimes (location, time, category, police unit identifier) and geographic boundaries of cities in the Los Angeles County. Given this dataset, we need to find the types of crimes that occur together. We will analyze such patterns for the cities Compton and West Hollywood.

There are a few crimes that are not spatial because their location is unknown or can not be identified. For some crimes like NSF [2] check frauds, the location is often not reported since it is irrelevant for the investigation purposes. We will filter out such crimes using filter function as shown in figure 4. Now that we only have the spatial crimes, we need to find the crimes that are located inside cities Compton and West Hollywood. To do this, we first filter out the said cities by name from the 'cities' collection and then use the contains operator to perform a spatial join between cities and crimes (see the first entry in table 1). We find 7,289 cases in Compton and 2,839 cases in West Hollywood.

The next step is to find the sets of crimes that occur near each other within a short span of time. To do this, we use the reccursive algorithm wherein we progressively seach specified number of meters around every crime to find crimes that occur within specified number of hours after the previous crime. We then run the same function reccursively on every crime selected by above procedure until we no longer find any crimes nearby. Also, we initially sort all the crimes in the ascending order of their time of occurance so that we can only look for crimes occuring in after the given crimes and not the ones before. To avoid same crime being counted multiple times, we set 'seen' flag inside the crime object every time the crime is passed to the function. All the crimes that are already 'seen' are filtered out from the list of nearby crimes before performing any further process. Algorithm 1 shows the procedure for searching crimes that are spatiotemporally nearby the given crime. Algorithm 2 shows the procedure to calculate the series of nearby crimes starting from the given crime and moving forward in the time. Algorithm 2 can be applied on every crime after temporally sorting crimes in ascending order. Result will be the set of all the series of nearby crimes. We found 461 such series from Compton and 117 series from West Hollywood.

[2]Not Sufficient Funds

Data: *point*: A point corresponding to a crime,
dist_th: Distance threshold in meters,
time_th: Time threshold in seconds
Result: All crimes spatiotemporally nearby to *point*
if *point is seen* **then**
  return empty list;
end
mark *point* as seen;
searchSpace := buffer of dist_th around point;
crimes := crimes **containedBy** *searchSpace*;
results := *crimes* **within** [*point.time*, *point.time* + *time_th*];
return *results*;

Algorithm 1: SEARCH_NEARBY, Finding crimes spatiotemporally nearby to a given point

Data: *point*: A point corresponding to a crime,
dist_th: Distance threshold in meters,
time_th: Time threshold in seconds,
series: current series; initially empty
Result: Series of crimes starting with *point*
nearby := **SEARCH_NEARBY** (*point*, *dist_th*, *time_th*);
if *nearby is empty list* **then** return *series*;
;
foreach *nearby_crime in nearby* **do**
  if *nearby_crime is not seen* **then**
    add *nearby_crime* to *series*;
    return **FIND_SERIES** (*nearby_crime*, *dist_th*, *time_th*, *series*);
  end
end

Algorithm 2: FIND_SERIES, Find series of crimes starting with current one

Once we find all such sets from our sample space, the next step is to find the types of crime that occur together most frequently in a set. For this, we use the apriori algorithm [**apriori**] for the association mining. The basic idea of apriori is that any association can not be stronger than the strength of its weakest subset. We use this idea to significantly reduce the number of comparisons necessary to find all the relevant associations which in this case are cooccurances of crime categories. We provide apriori algrithm as an extension to our tool which makes it very easy to perform such analyses from right inside the tool. Table 4 shows the most commonly associated crime types in spatiotemporally nearby crimes. We have included the top 5 association rules with highest support for both cities. Support indicates the frequency with which these rules occur in the population. Therefore, a support of 30% means the rule is observed in 30% of all the series found in that city. All the categories used are exactly as reported by the agencies. Figure 14 shows the code a user would have to write to solve the entire problem described in section 2 including pattern detection and apriori association mining. Note the significant conciseness of the code for a considerably complex operation. More importantly, the code can be written ad-hoc on a try and error basis.

6. EVALUATION

With respect to solving problem described in section 2, we will analyze the runtime performance of our tool with

Compton	
Association Rule	**Support**
Assault, Aggravated Assault, Vehicle/Boating Laws	23.86%
Theft/Larceny, Vehicle break-in/Theft, Assault	22.13%
Assault, Grand Theft Auto, Motor Vehicle Theft	20.61%
Non-aggravated assault, Vandalism, Narcotics	17.14%
Narcotics, Vehicle/Boating Laws, Assault	16.05%

West Hollywood	
Association Rule	**Support**
Vehicle/Boating Laws, Narcotics, Non-aggravated Assault	29.91%
Drunk/Alcohol/Drugs, Theft/Larceny, Vehicle/Boating Laws	26.49%
Vehicle/Boating Laws, Aggravated Assault, Non-aggravated Assault	22.22%
Vehicle/Boating Laws, Narcotics, Non-aggravated Assault	17.63%
Aggravated Assault, Narcotics, Non-aggravated Assault	15.38%

Table 1: Association rules for spatiotemporally associated crimes from Compton and West Hollywood with their support

respect to some other tools. We have divided this process into two parts: spatial joins and recursive pattern discovery. We compare first part with PostGIS and second part with the spatial extension of R language. All tests were performed on 2.3 GHz Intel i7 quad core machine with 8 gigabytes of DDR3 main memory running Mac OS X 10.9.3 (Mavericks).

6.1 Spatial Join

In section 5, we found crimes inside Compton and West Hollywood using spatial join between filtered 'city' and 'crime' collections. For the purpose of evaluation, we will take the spatial join between all cities and crimes. Collection 'city' has 250 polygons and collection 'crime' has 148,638 points. We compare the runtime for the said join using brute force approach (unindexed join), brute force but filtered by minimum bounding rectangle (MBR) approach, indexed join with PostGIS, single threaded indexed join with our tool and finally multithreaded indexed join with our tool. Figure 12 shows the runtimes for the join. All the times for indexed joins include time for the index creation. Note the great improvement caused over the brute force because of all the optimizations. Also, even though PogtGIS performed slightly better than our tool running on single thread, optimized multiprocessing outperforms everything else on the chart.

6.2 Recursive Pattern Discovery

In this section, we compare the performance of our tool with R language in running the algorithm for finding series of spatiotemporally nearby crimes (see algorithm 2: FIND_SERIES). Note that this is fairly complex since it potentially issues n^2 **contains** operations (**over** operations in R) during its runtime. It also issues **within** on the results of all the **contains** operations. Figure 13 shows runtime comparison between both tools for Compton and West Hollywood. Note that our tool performs better in both cases. This is mainly possible because of the spatial and temporal indexing.

Figure 8: Performance analysis of our tool against R in running the algorithm to extract series of spatiotemporally nearby crimes

```
var apriori = #IMPORT APRIORI
#coffee#
searchNear = (point, meters, hours)->
  return [] if point.seen
  point.seen = 1
  crimes.containedBy(point.buffer(meters), 'sp', (x)->1)
  .filter((x)-> x.properties.sp)
  .within(point.temp_env(hours, 1), 'time', (x)->1)
  .filter((x)-> x.properties.time)

series = (point, meters, hours, _series)->
  nearby = searchNear point, meters, hours
  return _series if nearby.length == 0
  nearby.forEach (near)->
    series near, meters, hours, _series.push(near)

apriori.init (crimes.map (crime)-> series(crime, 100,
     48)), 15
outtext apriori.run()
```

Figure 9: Code for solving problem from section 2

7. RELATED WORK

GIS systems allow representation of large data from multiple sources as layers. They come with some advanced functionality implementations like statistics, spatial joins etc. They also work with the raster data. Our tool allows users to perform most of the GIS tasks in a much more flexible programmatic way which allows to achieve much higher complexity of operations.

On the query side, SQL has spatial extension that provides functions to deal with data stored in a spatial database that follows the Object-Relational schema. Such spatial databases provide highly optimized operations and spatial indexing. When the data is too large to be processed with just one server, there are tools that help process such 'big' spatial data over Hadoop, which is a distributed execution framework. One good example of such languages if Hadoop GIS[**hgis**]. However, note that since these tools work on MapReduce framework, they do not offer interactive analysis. Their operation usually includes batch processing of large quantity of data stored in the Hadoop filesystem to obtain aggregation reports using mapreduce functions. Since our tool is designed for quick interactive analysis, it uses indexing and performs operations in parallel which makes it both fast and interactive. However, it should be noted that our tool is an in-memory tool which restricts the amount of data it can work with.

An analysis tool closer to our tool is the spatial modules for R language[**rspatial**]. This plugin lets users use a lot of advanced statistical capabilities of R for spatial objects. It also provides some basic visualizations using the 'plot' func-

Figure 7: Performance analysis of various tools and methods of execution for spatial join between collection 'crime' and 'city'

tion. R works both with vector and raster data. Our programming model is based on very simplistic data structures and intuitive operations. Moreover, since our tool is exclusively built for the spatiotemporal operations, it is better optimized for efficient execution of all common spatiotemporal operations.

8. CONCLUSION AND FUTURE SCOPE

We present a novel tool that helps make quick interactive spatiotemporal queries on the data in many different formats. The tool helps the analysis by providing quick aggregations, visualizations and mapping. Any advanced functionality may be added to this tool by writing extensions. It is very easy to include any existing JavaScript or CoffeeScript implementation as an extension to our tool. The tool makes operations faster by splitting them and running in parallel on multiple processor cores. We utilize most of available memory to improve speeds with thread-cacheing. The entire complexity of multiprocessing as well as synchronous-asynchronous operations is completely transparent to the users. Users can write any type of statements in desired sequence and expect a correct result.

We show the usefulness of our tool by demostrating its use on LA crime pattern analysis problem described in section 2 and further compare the process with some well-known tools in terms of performance. The results clearly indicate that our tool performs better in almost all the steps of execution. Moreover, it provides helpful visualizations and aggregations that are not available readily with the other tools making it a better problem solving utility.

In the future, we would like to increase the scale of this tool by running it on multiple machines simultaneously over the cloud. We would also like to explore in-memory cacheing systems like Memcached or Redis as our cacheing option. As of now, our tool only works with vector data. In future, we would like to add support for raster data.