ParkingIQ: A distributed parking location finder application

Aditya Sawhney, Akash Agrawal, MuraliKrishna Nallamothu and Srinivas Panchapakesan
University of Colorado at Boulder

{aditya.sawhney, akash.agrawal, murali.nallamothu, srinivas.panchapakesan}@colorado.edu

Abstract

In the grand scheme of things, the idea is to develop an automated Parking Management System which is geared towards streamlining, cost and efficiency wise, the parking management aspects (both from end user and government perspective) by developing intelligent systems which harnesses the power of distributed computing coupled with advancements in mobile/Smartphone computing. The main concept is that all the parking meters would constantly feed in their state information (vehicle is parked or not etc) to the central management cluster (CMC). Likewise, user information (park at given spot for 2 hrs, add time etc) would be fed into CMC. The CMC would in turn use this large dataset for various distributed computations like find all available parking spots in given vicinity (on demand request) or find all the violators in given vicinity (can be scheduled or on demand).

General Terms *Parking Solutions, Smartphone devices*

Keywords *Distributed Computing, Algorithms, REST+JSON Web services, Mule ESB, Apache Cassandra, Amazon EC2*

1. Introduction

ParkingIQ™ is a solution aimed at providing an intelligent and efficient solution for both users and enforcement agencies for utilizing parking services. The focus of the project was aimed at developing a pilot project which could be implemented at Parking and Transportation Services (PTS), University of Colorado, Boulder to enable users to primarily ascertain:

1. Nearest parking Lot
2. Number of the available spaces in the particular parking lot

Another important goal was to develop some sort of graphical reporting tool in order to aid PTS in event planning and estimation/analysis pertaining to parking space availability over a long period of time (e.g. quarter).

As a part of this objective, 'real' parking data from PTS was collected and stored in the database of the CMC. The architecture was developed to query the database for valid results and display them to the user. The client chosen for the project was a smart phone using the Android platform (compatible with Android 2.1 and below).

.

The same system can be implemented in any location which can provide

1. Real-time parking data
2. Access to parking Lot information.

The tools used in the project, which are explained in the later sections of the paper, are all capable of being configured as a distributed system with an emphasis on Scalability, High-Availability and eventual consistency.

Also, the roadmap for future additions to the functionality like

1. User should be able to 'purchase' a parking ticket using the client application. (currently Android)
2. Enforcement agencies should be able to locate 'violators' automatically without the need to manually scan through license-plates and parking meters.

have been incorporated to be included into the design during later stages.

All of the components chosen are open-source which provides a platform for the developer community to provide additions as well.

2. Architecture

In this section we introduce few of the key components of the system and provide a general overview of the component followed with the rationale behind using it and how it fits into the whole picture.

2.1 Apache Cassandra

Cassandra is an open source distributed database management system. It is an Apache Software Foundation top-level project designed to handle very large amounts of data spread out across many commodity servers while providing a highly available service with no single point of failure. It is a NoSQL solution that was initially developed by Facebook and powers their Inbox Search feature. Cassandra can be described as a BigTable data model running on an Amazon Dynamo-like infrastructure. Cassandra provides a structured key-value store with tunable consistency. Keys map to multiple values, which are grouped into column families. The column families are fixed when a Cassandra database is created, but columns can be added to a family at any time. Furthermore, columns are added only to specified keys, so different keys can have different numbers of columns in any given family. The values from a column family for each key are stored together. This makes Cassandra a hybrid data management system between a column-oriented DBMS (like BigTable) and a row-oriented store. Also, besides using the way of modeling of BigTa-

ble, it has properties like eventual consistency, the Gossip protocol, a master-master way of serving the read and write requests that are inspired by Amazon's Dynamo.

Data Model

The Cassandra data model is designed for distributed data on a very large scale. It trades ACID-compliant data practices for important advantages in performance, availability, and operational manageability. The basic concepts in cassandra are:

- Cluster: the machines (nodes) in a logical Cassandra instance. Clusters can contain multiple keyspaces.
- Keyspace: a namespace for ColumnFamilies, typically one per application.
- ColumnFamilies contain multiple columns, each of which has a name, value, and a timestamp, and which are referenced by row keys.
- SuperColumns can be thought of as columns that themselves have subcolumns.

The column is the lowest/smallest increment of data. It's a tuple (triplet) that contains a name, a value and a timestamp. A column family is a container for columns, analogous to the table in a relational system. You define column families in your storage-conf.xml file, and cannot modify them (or add new column families) without restarting your Cassandra process. A column family holds an ordered list of columns, which you can reference by the column name. Column families have a configurable ordering applied to the columns within each row, which affects the behavior of the get_slice call in the thrift API. Out of the box ordering implementations include ASCII, UTF-8, Long, and UUID (lexical or time). In Cassandra, each column family is stored in a separate file, and the file is sorted in row (i.e. key) major order. Related columns, those that you'll access together, should be kept within the same column family. The row key is what determines what machine data is stored on. Thus, for each key you can have data from multiple column families associated with it. However, these are logically distinct, which is why the Thrift interface is oriented around accessing one ColumnFamily per key at a time. A keyspace is the first dimension of the Cassandra hash, and is the container for column families. Keyspaces are of roughly the same granularity as a schema or database (i.e. a logical collection of tables) in the RDBMS world. They are the configuration and management point for column families, and is also the structure on which batch inserts are applied. Cassandra also supports super columns: columns whose values are columns; that is, a super column is a (sorted) associative array of columns. One can thus think of columns and super columns in terms of maps: A row in a regular column family is basically a sorted map of column names to column values; a row in a super column family is a sorted map of super column names to maps of column names to column values. Cassandra supports pluggable partitioning schemes with a relatively small amount of code. Out of the box, Cassandra provides the hash-based RandomPartitioner and an OrderPreservingPartitioner. RandomPartitioner gives you pretty good load balancing with no further work required. OrderPreservingPartitioner on the other hand lets you perform range queries on the keys you have stored, but requires choosing node tokens carefully or active load balancing. Systems that only support hash-based partitioning cannot perform range queries efficiently. Unlike with relational systems, where you model entities and relationships and then just add indexes to support whatever queries become necessary, with Cassandra you need to think about what queries you want to support efficiently ahead of time, and model appropriately. Since there are no automatically-provided indexes, you will be much closer to one ColumnFamily per query than you would have been with tables:queries relationally. Cassandra is much, much faster at writes than relational systems, without giving up speed on reads.

A table in Cassandra is a distributed multidimensional map indexed by a key. The value is an object that is highly structured. The row key in a table is a string with no size restrictions, although typically 16 to 36 bytes long. Every operation under a single row key is atomic per replica no matter how many columns are being read or written into. Cassandra can handle maps with four or five dimensions:

Map with four dimensions:

- Keyspace -> Column Family
- Column Family -> Column Family Row
- Column Family Row -> Columns
- Column -> Data value

Map with five dimensions:

- Keyspace -> Super Column Family
- Super Column Family -> Super Column Family Row
- Super Column Family Row -> Super Columns
- Super Column -> Columns
- Column -> Data value

Applications can specify the sort order of columns within a Super Column or Simple Column family. The system allows columns to be sorted either by time or by name. Time sorting of columns is exploited by applications like Facebook Inbox Search, where the results are always displayed in time-sorted order. Any column within a column family is accessed using the convention column_family : column, and any column within a column family that is of type super is accessed using the convention column_family : super_column : column. When the cluster for Apache Cassandra is designed, an important point is to select the right partitioner. Two partitioners exist

- RandomPartitioner (RP): This partitioner randomly distributes the key-value pairs over the network, resulting in a good load balancing. Compared to OPP, more nodes have to be accessed to get a number of keys.
- OrderPreservingPartitioner (OPP): This partitioner distributes the key-value pairs in a natural way so that similar keys are not far away. The advantage is that fewer nodes have to be accessed. The drawback is the uneven distribution of the key-value pairs.

.

Features

- *Decentralized*: Every node in the cluster has the same role. There is no single point of failure. Data is distributed across the cluster (so each node contains different data), but there is no master as every node can service any request.
- *Elasticity*: Read and write throughput both increase linearly as new machines are added, with no downtime or interruption to applications.
- *Fault-tolerant*: Data is automatically replicated to multiple nodes for fault-tolerance. Replication across multiple data centers is supported. Failed nodes can be replaced with no downtime.
- *Tunable consistency*: Writes and reads offer a tunable level of consistency, all the way from "writes never fail" to "block for all replicas to be readable", with the quorum level in the middle.

Limitations

These are the major limitations one must take into account while designing a model

- All data for a single row must fit (on disk) on a single machine in the cluster. Because row keys alone are used to determine the nodes responsible for replicating their data, the amount of data associated with a single key has this upper bound.
- A single column value may not be larger than 2GB.
- The maximum of column per row is 2 billion.

Users of Cassandra

- Cisco's WebEx uses Cassandra to store user feed and activity in near real time.
- Cloudkick uses Cassandra to store the server metrics of their users.
- Digg, a large social news website, announced on Sep 9th, 2009 that it is rolling out its use of Cassandra and confirmed this on March 8, 2010. TechCrunch has since linked Cassandra to Digg v4 reliability criticisms and recent company struggles. Lead engineers at Digg later rebuked these criticisms as red herring and blamed a lack of load testing.
- Facebook used Cassandra to power Inbox Search, with over 200 nodes deployed. This was abandoned in late 2010.
- IBM has done research in building a scalable email system based on Cassandra.
- Rackspace is known to use Cassandra internally.
- Reddit switched to Cassandra from memcacheDB on March 12, 2010 and experienced some problems with overload handling in Cassandra in May.
- Twitter announced it is planning to use Cassandra because it can be run on large server clusters and is capable of taking in very large amounts of data at a time. Twitter continues to use it but not for Tweets themselves.
- Yakaz uses Cassandra on a five-node cluster to store millions of images as well as its social data.

2.2 Amazon EC2

The web service and database of ParkingIQ needs to be distributed world wide as it is a distributed system. It requires multiple servers serving client requests all over the world. Thus the deployment of ParkingIQ needs to be highly scalable on multiple server machines as well as distributed storage. Amazon provides various web services for such requirements called the Amazon Web Services (AWS).

AWS

Since early 2006, Amazon Web Services (AWS) has provided companies of all sizes with an infrastructure web services platform in the cloud.

With AWS, customer can requisition compute power, storage, and other services–gaining access to a suite of elastic IT infrastructure services as per business demands. With AWS customers have the flexibility to choose whichever development platform or programming model makes the most sense for the problems they're trying to solve. Customer pay only for what they use, with no upfront expenses or long-term commitments, making AWS the most cost-effective way to deliver their application to their own customers and clients. And, with AWS, one can take advantage of Amazon.com's global computing infrastructure that is the backbone of Amazon.com's multi-billion retail business and transactional enterprise whose scalable, reliable, and secure distributed computing infrastructure has been honed for over a decade.
Using Amazon Web Services, an e-commerce web site can weather unforeseen demand with ease; a pharmaceutical company can "rent" computing power to execute large-scale simulations; a media company can serve unlimited videos, music, and more; and an enterprise can deploy bandwidth-consuming services and training to its mobile workforce.

Amazon EC2

Amazon Elastic Compute Cloud (Amazon EC2) is a web service that provides resizable compute capacity in the cloud. It is designed to make web-scale computing easier for developers.

Amazon EC2's simple web service interface allows customers to obtain and configure capacity with minimal friction. It provides them with complete control of their computing resources and lets them run on Amazon's proven computing environment. Amazon EC2 reduces the time required to obtain and boot new server instances to minutes, allowing customers to quickly scale capacity, both up and down, as their computing requirements change. Amazon EC2 changes the economics of computing by allowing them to pay only for capacity that they actually use. Amazon EC2 provides developers the tools to build failure resilient applications and isolate themselves from common failure scenarios.

EC2 Functionality

Amazon EC2 presents a true virtual computing environment, allowing customer to use web service interfaces to launch instances with a variety of operating systems, load them with their custom application environment, manage network's access permissions, and run image using as many or few systems as desired.
To use Amazon EC2, simply:

- Select a pre-configured, template image to get up and running immediately. Or create an Amazon Machine Image

(AMI) containing your applications, libraries, data, and associated configuration settings.

- Configure security and network access on your Amazon EC2 instance.
- Choose which instance type(s) and operating system you want, then start, terminate, and monitor as many instances of your AMI as needed, using the web service APIs or the variety of management tools provided.
- Determine whether you want to run in multiple locations, utilize static IP endpoints, or attach persistent block storage to your instances.
- Pay only for the resources that you actually consume, like instance-hours or data transfer.

Features

Amazon EC2 provides a number of powerful features for building scalable, failure resilient, enterprise class applications, including:

Amazon Elastic Block Store – Amazon Elastic Block Store (EBS) offers persistent storage for Amazon EC2 instances. Amazon EBS volumes provide off-instance storage that persists independently from the life of an instance. Amazon EBS volumes are highly available, highly reliable volumes that can be leveraged as an Amazon EC2 instance's boot partition or attached to a running Amazon EC2 instance as a standard block device.

Multiple Locations – Amazon EC2 provides the ability to place instances in multiple locations. Amazon EC2 locations are composed of Regions and Availability Zones.

Elastic IP Addresses – Elastic IP addresses are static IP addresses designed for dynamic cloud computing. An Elastic IP address is associated with your account not a particular instance, and you control that address until you choose to explicitly release it.

Amazon Virtual Private Cloud – Amazon VPC is a secure and seamless bridge between a company's existing IT infrastructure and the AWS cloud.

Elastic Load Balancing – Elastic Load Balancing automatically distributes incoming application traffic across multiple Amazon EC2 instances. It enables you to achieve even greater fault tolerance in your applications, seamlessly providing the amount of load balancing capacity needed in response to incoming application traffic.

High Performance Computing (HPC) Clusters – Customers with complex computational workloads such as tightly coupled parallel processes, or with applications sensitive to network performance, can achieve the same high compute and network performance provided by custom-built infrastructure while benefiting from the elasticity, flexibility and cost advantages of Amazon EC2.

Auto Scaling – Auto Scaling allows you to automatically scale your Amazon EC2 capacity up or down according to conditions you define.

2.3. Mule Enterprise Service Bus (ESB)

The ParkingIQ system needs a backend web server running to serve client requests for available parking spaces. To fulfill those requests, there is a need for services to convert or analyze clients current coordinates, look for available nearby parking lots, there present status (current available spaces in each parking lots). This also requires us to maintain a huge database and an ability of services to query those databases for required information. All of these services (serve clients, locate coordinates, find parking spots, update database periodically) need to work together by passing messages or requests between each other. We use the Mule Enterprise Service Bus (ESB) to meet our requirements. Most of our services are running in Java with Mule transports and connectors to link those services together.

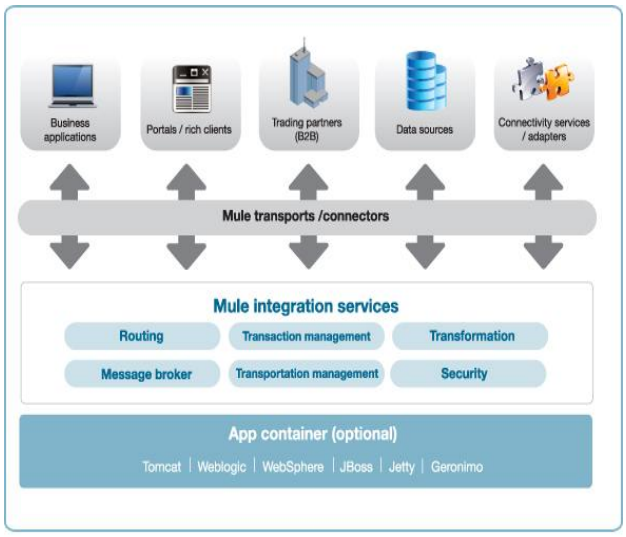
Mule ESB is a lightweight Java-based enterprise service bus (ESB) and integration platform that allows developers to connect applications together quickly and easily, enabling them to exchange data. Mule ESB enables easy integration of existing systems, regardless of the different technologies that the applications use, including JMS, Web Services, JDBC, HTTP, and more.
The key advantage of an ESB is that it allows different applications to communicate with each other by acting as a transit system for carrying data between applications within customers' enterprise or across the Internet.

Mule ESB includes powerful capabilities that include:

- **Service creation and hosting** — expose and host reusable services, using Mule ESB as a lightweight service container

.

- **Service mediation** — shield services from message formats and protocols, separate business logic from messaging, and enable location-independent service calls
- **Message routing** — route, filter, aggregate, and re-sequence messages based on content and rules
- **Data transformation** — exchange data across varying formats and transport protocols



Advantages of Mule:
Mule ESB is lightweight but highly scalable, allowing programmer to start small and connect more applications over time. Mule manages all the interactions between applications and components transparently, regardless of whether they exist in the same virtual machine or over the Internet, and regardless of the underlying transport protocol used.
There are currently several commercial ESB implementations on the market. However, many of these provide limited functionality or are built on top of an existing application server or messaging server, locking programmers into that specific vendor. Mule is vendor-neutral, so different vendor implementations can plug in to it.

Mule provides many advantages over competitors, including:

- Mule components can be any type needed. One can easily integrate anything from a "plain old Java object" (POJO) to a component from another framework.
- Mule and the ESB model enable significant component reuse. Unlike other frameworks, Mule allows you to use your existing components without any changes. Components do not require any Mule-specific code to run in Mule, and there is no programmatic API required. The business logic is kept completely separate from the messaging logic.

.

- Messages can be in any format from SOAP to binary image files. Mule does not force any design constraints on the architect, such as XML messaging or WSDL service contracts.
- You can deploy Mule in a variety of topologies, not just ESB. Because it is lightweight and embeddable, Mule can dramatically decrease time to market and increases productivity for projects to provide secure, scalable applications that are adaptive to change and can scale up or down as needed.

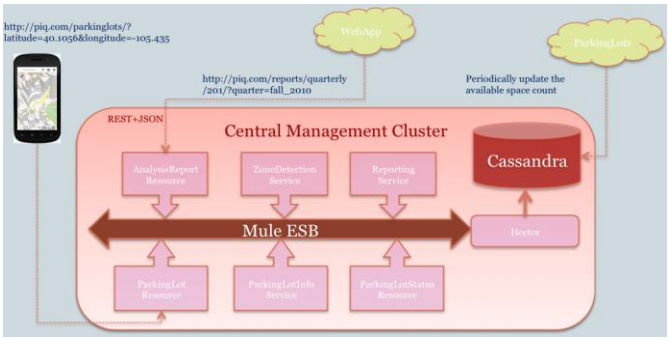
3. Design Overview

The main design goals of the system are as follows:

1. **Elastic Scalability:** The system should be able to scale up or down based on demand at multiple levels. More specifically, the system should be deployable at AmazonEC2 so that more infrastructure/hardware can be easily added, the system should work with Hadoop so that sudden surge in computational demands can be easily met, the data store should be capable of storing and processing large scale data with no visible impact in latency.
2. **High Availability:** The system should be failover tolerant and have some level of data replication. It should be guaranteed that writes will always be available otherwise it will impact the perceived reliability of data.
3. **Eventual Consistency:** It is acceptable to compromise on the consistency on the reads. In particular, it is not a big deal if sometimes we report incorrect space availability.
4. **Open Source Components:** Due to cost constraints we want to use open source components as much as possible.

As mentioned before, the server comprises of Mule ESB as the backbone with a bunch of services hanging off it. Mule provides a pluggable architecture where in additional components can be seamlessly deployed. In addition to the base deployment of Mule we use Jersey component for exposing REST style resources. In the current implementation there are two main resources, ParkingLotResource and AnalysisReportResource which support HTTP GET operations for parking lot information and quarterly analysis report respectively. Also, we rely on Jackson component for generating response in JSON format. We decided to use JSON as an output format for our services because it lightweight and highly inter-operable data format. The main point to note is that Mule serves as an application server (gateway for receiving requests) and an integration platform for orchestration of various services involved.

In order to store the data we decided to use Apache Cassandra which is an open source implementation provided by Facebook based on the 'No SQL' paradigm. Cassandra is basically a hybrid of Amazon's Dynamo and Google's BigTable implementations. The main reason for using Cassandra is that it not only meets all the design goals, but, also integrates well with AmazonEC2 and Apache Hadoop/MapReduce. In addition, we use Hector (client side libraries for Cassandra) for interfacing with Cassandra and executing various queries.



4. Implementation

4.1 Server Components

The server comprises mainly of two set of components:

1. REST Resources
2. Modular Services

The REST resources are basically Java classes which are geared towards handling the various HTTP requests. It is responsible for extracting out the param and query parameters from the URI then use the services to get the intended work done, compose the response in desired format (which is JSON in our case) and send it back to the client. Currently, there are two resources:

a. **ParkingLotResource:** Supports retrieving a list of parking lots (along with location and available space count data) which are in the vicinity of a given location.

b. **AnalysisReportResource:** Supports retrieving the quarterly report (daily average available space) for given quarter and lot.

The service components are designed to be modular so that it can be re-used in different contexts. Currently, there are following services:

a. **ZoneDetectionService**: This service is responsible to evaluate the zone to which the given location (latitude and longitudes) belongs.

b. **ParkingLotInfoService**: This service is responsible for retrieving static information (lot id, type, location) pertaining to parking lots based on a given filtering criteria. It currently supports two filtering criteria: for a particular lot id and all lots which belong to given zone. The service internally queries Cassandra for retrieving this information.

c. **ParkingLotStatusService**: This service is responsible for retrieving the current status of the parking lots. That is, currently available spaces. Again, this service internally queries Cassandra for retrieving this information.

d. **ParkingLotReportingService**: This service is responsible for retrieving the quarterly report for given parking lot and quarter.

.

Following is the code snippet of the ParkingLotResource for illustration purposes:

```
/**
 * Represents the REST resource for parking lots.
 */
@Path("/parkinglots")
public class ParkingLotResource {

/**
 * Get parking lot information for given location.
 *
 * @param latitude the latitude of location
 * @param longitude the longitude of location
 * @return the list of parking lots which are close to the given location
 */
  @GET
  @Produces("application/json")
  public List<ParkingLotInfo> getParkingLots(
  @DefaultValue("0.0") @QueryParam(Const.Param.LATITUDE) float latitude,
  @DefaultValue("0.0") @QueryParam(Const.Param.LONGITUDE) float longitude) {
    GeoPoint location = new GeoPoint(latitude, longitude);

    // Figure out the zone to which the given belongs
    Zone zone = this.zoneDetectionService.identifyZone(location);

    // Get all the parking lots which lie in the zone
    List<ParkingLotInfo> parkingLots = this.parkingLotInfoService.getParkingLotInfo(zone);

    // Get the current status (available spaces) for each of those lots
    this.parkingLotStatusService.updateParkingLotStatus(parkingLots);

    return parkingLots;
  }
}
```

The current incarnation of the server supports the following workflows:

Find parking lots

*Input:*Current location – latitude and longitude

*Output:*List of parking lots (with location and available space count) in JSON format

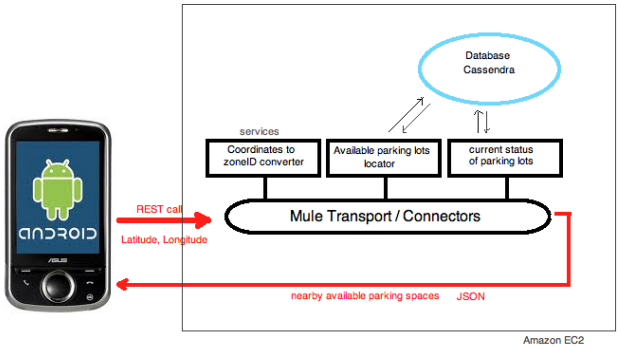
Flow:

a. Use ZoneDetectionService to identify the zone to which the given location belongs.

b. Retrieve a list of all the parking lots which are in the above zone using ParkingLotInfoService.

c. For each of those parkinglots retrieve the current staus (available spaces) using ParkingLotStatusService.

d. Compile and return the results in JSON format.



Get quarterly report

Input: Lot id and quarter for which report is required

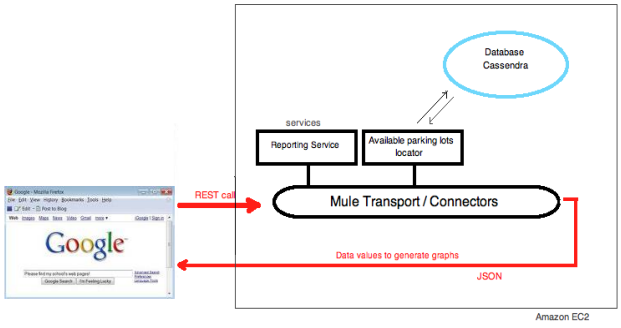
*Output:*Corresponding report with average space availability from Monday to Friday for three sessions a day, i.e. morning, afternoon and evening.

Flow:

a. Use the ParkingLotReportingService to retrieve the quarterly report from Cassandra.

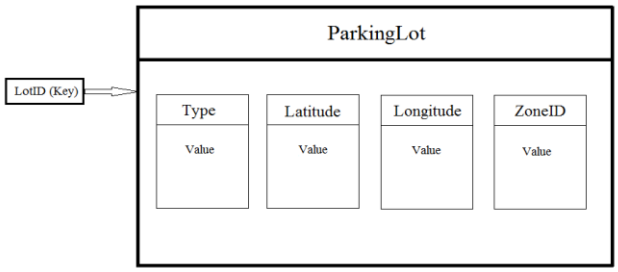
b. Retrieve the information about the parking lot using ParkingLotInfoService.

c. Compile those two results into a single artifact and return the result in JSON format.



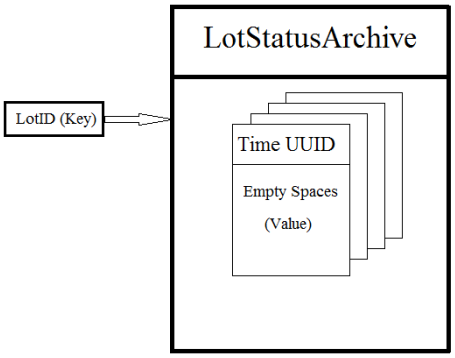
4.2 Database

We have used Cassandra as our database management system. The project required us to maintain three separate tables to manage all out data. The first table was named ParkingLot and was used to store all the static information about each parking lot. A pictorial representation of this table is shown in the figure below:

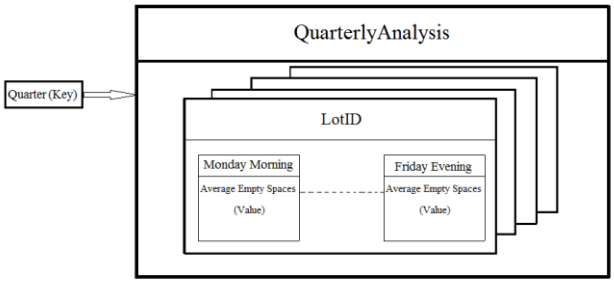


In the above table all the static information regarding each parking lot are stored as columns. The type, latitude, longitude and ZoneID are grouped together to form the ColumnFamily, ParkingLot.

The second table is created to store all the dynamic information about each parking lot as and when we receive them. The information is available empty spaces in each lot. In this table the time is used as the column and its corresponding value is recorded. Every time a new reading is recorded, a new column is created. Here the time UUID acts as a comparator. This can be used to sort the database in order to get the latest number of empty slots. The columns in this column family expands as an when we get new data.



The third table which we are using is generated by doing a MapReduce operating on the LotStatusArchive table. This table is used to store a time wise summary of average empty spaces in a parking lot over a quarter. The CU Parking services break the data down into secession of morning, afternoon and evening during weekdays. We have created our database to stay compatible with their system. The figure below represents this database.



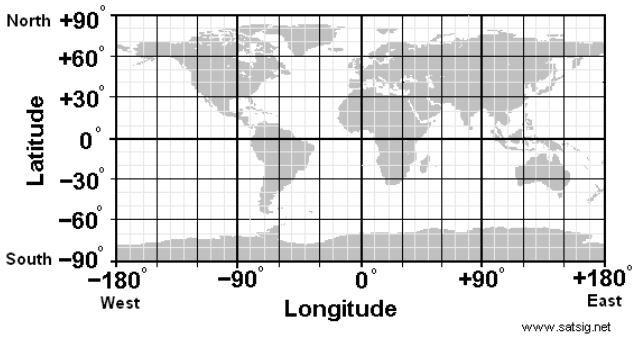
We have modeled the database by creating a column for every session for all week days. All these columns are specific to each lot, thus LotID is a Super Column. Given a Quarter say fall 2010 and specifying a parking lot we can get the average empty spaces for each lot at different times of the day.

4.3 Division of ZoneID

For improving the efficiency of database search and also to speed up the data query process, cities are divided into geographical zones based on the geo coordinates (latitudes and longitudes). Whenever a client wants available parking spots near to his present or current location, the ParkingIQ system determines the Zone in which the client is present and queries for all the available parking lots in that particular Zone.

Although a better and more accurate way is to get the clients current location and find all the available parking lots in a particular radius area. But such system will require a distance calculation of every single database entry of parking lots with the current coordinates, under which case its going to take a lot of time to do all the distance measurement and comparison and wont be a very efficient method for performing data query.

To overcome the above limitation and to improve the data query response as well as efficiency, we used a method of dividing the area into small blocks of walk able distance areas.



When a request arrives form the android client to the ParkingIQ server. It first reaches to the Coordinates to ZoneID converter service of Mule ESB. The service takes in the latitude and longitude value and multiplies it by 100. As per our calculation 1 degree at latitude 45degree is about 78.5 Km. Thus dividing by 100 we get around 785 meters. Similar logic is used for the longitude. Thus we divide the plane into Zones of walk able distance which also helps in faster database querying.

```
/**
 * Implementation of ZoneDetectionService interface.
 */
public class ZoneDetectionServiceImpl implements ZoneDetectionService {

	/* (non-Javadoc)
	 * @see edu.colorado.piq.service.ZoneDetectionService#identifyZone(edu.colorado.
	 */
	public Zone identifyZone(GeoPoint currentCoord) {
		String zoneId = null;

		if (currentCoord.isValid()) {
		int lat = (int)(currentCoord.getLatitude()  * 100.0);
		int lng = (int)(currentCoord.getLongitude() * 100.0);
			zoneId = String.format("%1d_%2d", lat, lng);
		}
		return new Zone(zoneId); // 40001_-105263(>1) 40005_-105270(=1)
	}
}
```

4.4 Android Client

The Android client application is used by customers to find out details about the nearest parking spot. The basic implementation in the project consists of:

1. Collect a location input from the customer as [latitude, longitude] and display the details of the nearest parking lots.
2. Use the current location of the customer [obtain from GPS] and display the details of the nearest parking spots.
    a. Add an overlay item which displays the current location of the user.
    b. Upon request, display the nearest parking lots [based on Zone ID] on the Android client.
3. [Optional] Use the place entered by the user to display the details of the parking lots.

.

Project settings

The android application has been developed to be compatible with Android 2.1 [and below]. The required parameters should be set in the android manifest.xml file:

1. INTERNET
2. ACCESS_COARSE_LOCATION
3. ACCESS_FINE_LOCATION

Also, as Google maps library is used, the library *com.google.android.maps* must be included in the AndroidManifest.xml.

GUI Outline

Each view in the Android application has been configured as an *Activity*, with the screen displaying the map configured as a *MapActivity*. Figure 1 explains the first screen of the Android application.

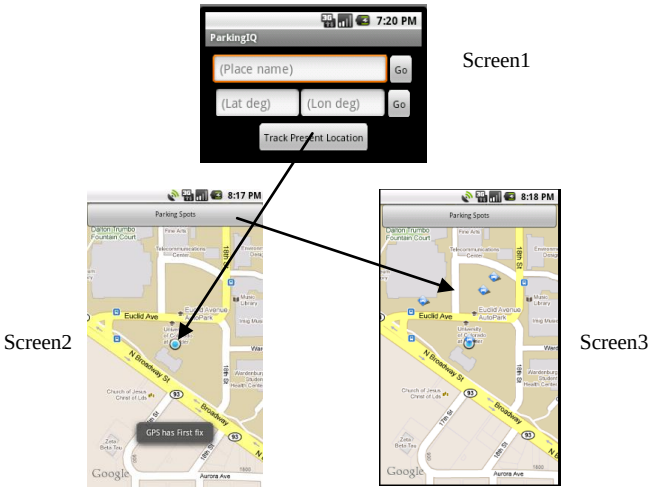


Figure 1 Screen: Track Present Location

The welcome screen (Screen 1) prompts the user to either enter the name of the place [optional-not fully implemented] or the exact values of the latitude and longitude or use the current location of the user. The actions are encoded with the button instance *Track Present Location* and *Go* to launch the next Activity, in this case Screen 2.

When *Track Present Location* button is clicked, the *onClickListener()* in the code triggers the next MapActivity and launches Screen 2 with an overlay item present on the map. In order to obtain the parking spots nearby, the user has to click the *Parking Spots* button which queries the Database through the Mule ESB server and provides details of the nearby parking lots

The functionality is almost the same when the user enters latitude and longitude values of interest. The *Go* button when clicked launches the *MapActivity* and the *Parking Spots* button provides the details of the parking Lots.

Google Maps Display

To implement the google maps API, there is a requirement to sign up for an API key. During development phases, the key present in the *debug.keystore* of the android-sdk installation would suffice. To register for the API key pair, the developer needs to sign with Google. This key is machine-specific for the development computer if a temporary key is obatined using the debug certificate. Thus, to develop on more than one machine, each will have a different *apiKey* and there is a need to change the entry in the **.xml* file to the appropriate one if the project is moved between machines.

Geocoding

Geocoding is the transformation of a location description such as a street address into a (latitude, longitude) coordinate.
This is used to transform the location in the first text-box to a *<latitude, longitude>* co-ordinate. The *getFromLocationName()* method is used to achieve this objective.

Present Location

The *MapActivity* implements the *LocationListener* class and the GPS Status package is used to lock the current location of the user.
The parameters of the *GpsStatus.Listener()* are set accordingly to ascertain the current location of the user:

1. GPS_EVENT_STARTED
2. GPS_EVENT_FIRST_FIX
3. GPS_EVENT_STOPPED
4. GPS_EVENT_SATELLITE_STATUS

An instance of *LocationManager* is not instantiated directly but rather using *getSystemService(Context.LOCATION_SERVICE)*. This uses the current application context and the GpsStatus Listener listens for changes in GPS updates. Android devices obtain precise locations from GPS fixes (ACCESS_FINE_LOCATION permission in AndroidManifest.xml), but they may offer less precise locations from triangulation on cell phone towers and proximity to WIFI hotspots (ACCESS_COARSE_LOCATION permission in AndroidManifest.xml).

5. Conclusions

With increasing number of automobiles on the road, it is becoming harder and harder to find empty parking spaces. Especially, in big cities such as New York and Chicago. We have described ParkingIQ: A distributed parking location finder application. A system used for finding empty parking located near our location. The front end of our system is an Android smart phone which a user can use to find empty parking lots near his location. The backend of our system is entirely distributed and data is stored in a cluster of computers. This system is currently implemented keeping in mind database and systems used by the CU Parking services. Our system is a prototype which is currently working on archived data provided to us by the CU parking services, but, can be easily expanded to accept real time inputs from the parking lots allowing us to track the live information of the parking lots. Currently, the CU Parking services only update their data base three times every day. In order for our system to be effective in densely populated cities we would need to update our data base in a matter of minutes. With the installation of proper hardware in the parking lots we can keep a live record of empty spaces in each parking lot and the system can be utilized effectively by the public.

Refer to the project website http://code.google.com/p/parking-iq/ for more details and access to the source code.

6. Future Work

As a future work to this project involve integrating our system with that of the Parking Services. With the installation of proper hardware sensors to keep a live record of number of empty parking spaces, the lives of Parking Services and customers can be made a lot easier.

Additional features which can be added to this project can be the use of a booking service. The customers can use this service from their android phones to pre book a parking space. We can even add a functionality to create a profile for each user and save their credit card information along with it and use it to book parking spaces.

A tool can be implemented on top of our system which is used to keep track of parking violations and provide this information to the enforcement agencies. This will help in automating the entire process of writing parking tickets.

Acknowledgments

We thank Prof Rick Han to help us through the entire project development phase and also for his excellent guidance and teaching. We would also like to thank the Parking and Transportation Services of University of Colorado at Boulder and especially Marta Williams for providing us with the dataset to test our system and for discussing with us about the present CU parking system which helped us a lot in designing our project.