San Francisco crime:

deep analysis and prediction model

Álvaro·López·Sánchez  
Julio·Sorroche·García  
Lupicinio·García·Ortiz

**Abstract**

En este proyecto utilizaremos la información de la actividades criminales producidas en la ciudad de San Francisco organizadas en un volúmen masivo de datos, y con el fin de seguir una serie de etapas de preprocesamiento, análisis para finalmente obtener un modelo de predicción. Para ello seguiremos diferentes perspectivas y metodologías a lo largo del almacenamiento, preprocesamiento y representación de la información con ayuda de MongoDb, Cassandra y Neo4j. Para estar mejor preparado para responder a cualquier actividad delictiva y predecirlos, es importante comprender los patrones para que se produzca un delito. Mediante varios modelos de clasificación y predicción basado en aprendizaje profundo supervisado y no supervisado; optaremos por la opción que se ajuste mejor para realizar predicciones.

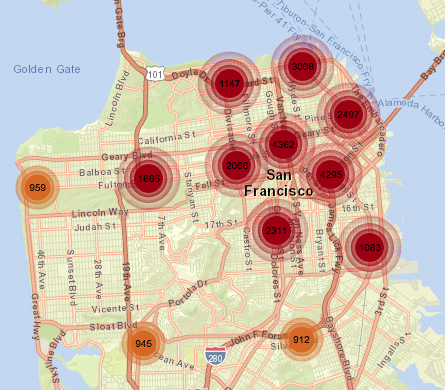
# Introducción

El objetivo principal del proyecto es clasificar la información de la actividad criminal de San Francisco dado el tiempo y la localización. Realizar un análisis y obtener un conjunto de modelos de representación de la información de éstas actividades; y que a partir de dichos modelos podamos consultar información con mejor enfoque y con valores más discretos. Para ello introduciremos las tecnologías MongoDb, Cassandra y Neo4j y obtener diferentes perspectivas de proceso y manejo de la información de forma respectiva.

En el resultado final la información disponible no solo se basa de la información en bruto obtenida a partir de un volumen masivo de datos correspondientes a incidencias desde el 2013 hasta la actualidad; sino que ademas al ajustar un modelo de predicción se podrá encontrar datos futuros; y la mejor manera es visualizar dichas predicciones sobre la información presente.

## I. Dataset

Nuestro dataset está publicamente disponible desde el portal del departamento Policial de *San Francisco*. Contiene datos de todas las incidencias generadas por actividades criminales producidas desde el año 2003 hasta la actualidad. De forma concreta, esta información se encuentra expuesta de forma pública desde el sistema de actualización diaria del *SFPD Crime Incident Reporting* (sistema de reportes del departamento policial) a traves de su plataforma Socrata.



La información esta representada por un conjunto de 2 millones de filas, que contiene información detallada de actividades criminales (o incidencias) representada por: *tipo de delito, resolución, distrito, coordenadas y dirección de la zona, nº de incidencia vinculada*; todos producidos entre el 2003 hasta la actualidad.

* **Dates -** fecha y hora del incidente del crimen
* **Category** - categoría del incidente criminal
* **Descript** - descripción detallada del incidente del crimen
* **Dayoftheweek** de la semana: el día si la semana
* **PdDistrict**- nombre del distrito del departamento de policía
* **Resolution**: cómo se resolvió el incidente del crimen
* **Address**: dirección aproximada del incidente delictivo
* X - longitud
* Y - latitud

En el caso de las **coordenadas** (x,y). La X y la Y dan esencialmente el parámetro de ubicación. Es un dato interesante que veremos aprovechar mediante tecnologia que proporciona **MongoDb** con respecto a geolocalizaciones.

La fecha se agrupa como una fecha, hora y día de la semana. Esto se divide para su uso posterior en las siguientes secciones.

150060275;NON-CRIMINAL;LOST PROPERTY;Monday;2015-01-19 14:00:00;19;2015;01;14;MISSION;NONE;18TH ST / VALENCIA ST;-122.42158168137;37.7617007179518;(37.7617007179518, -122.42158168137);15006027571000

150098210;ROBBERY;ROBBERY, BODILY FORCE;Sunday;2015-02-01 15:45:00;01;2015;02;15;TENDERLOIN;NONE;300 Block of LEAVENWORTH ST;-122.414406029855;37.7841907151119;(37.7841907151119, -122.414406029855);15009821003074

## II. Fases

A partir de la estructura de datos definida para el dataset se llevará a cabo una serie de procesos y fases, que nos permitan almacenar dicha información desde distintas perspectivas de almacenaje y representación masiva de datos

* **Mongodb**, como manejo de información a través de documentos
* **Cassandra**, con potencial residente en la arquitectura para el volumen de datos
* **Neo4j**, como orientación en la busqueda de relaciones entre los datos

# Contextualización del proyecto

## I. Antecedentes

.....

## II. Proyectos en la actualidad

## .....

# Fundamentos teóricos/prácticos

En este apartado inicialmente se discutiran las capacidades en las fases de preprocesamiento, análisis y estructura de la información de cada tecnología, conectores usados para realizar la representación de la información.

## Estructura y arquitectura de la información

En este apartado se realiza un desglose teórico de como cada modelo de base de datos correspondiente a las tecnologias usadas dispone la información, y de como la arquitectura se basa.

[I. Cassandra](https://github.com/orial/grupo_Spark/blob/master/Procesamiento/cassandra/readme.md#preprocesamiento-e-importaci%C3%B3n-de-datos)

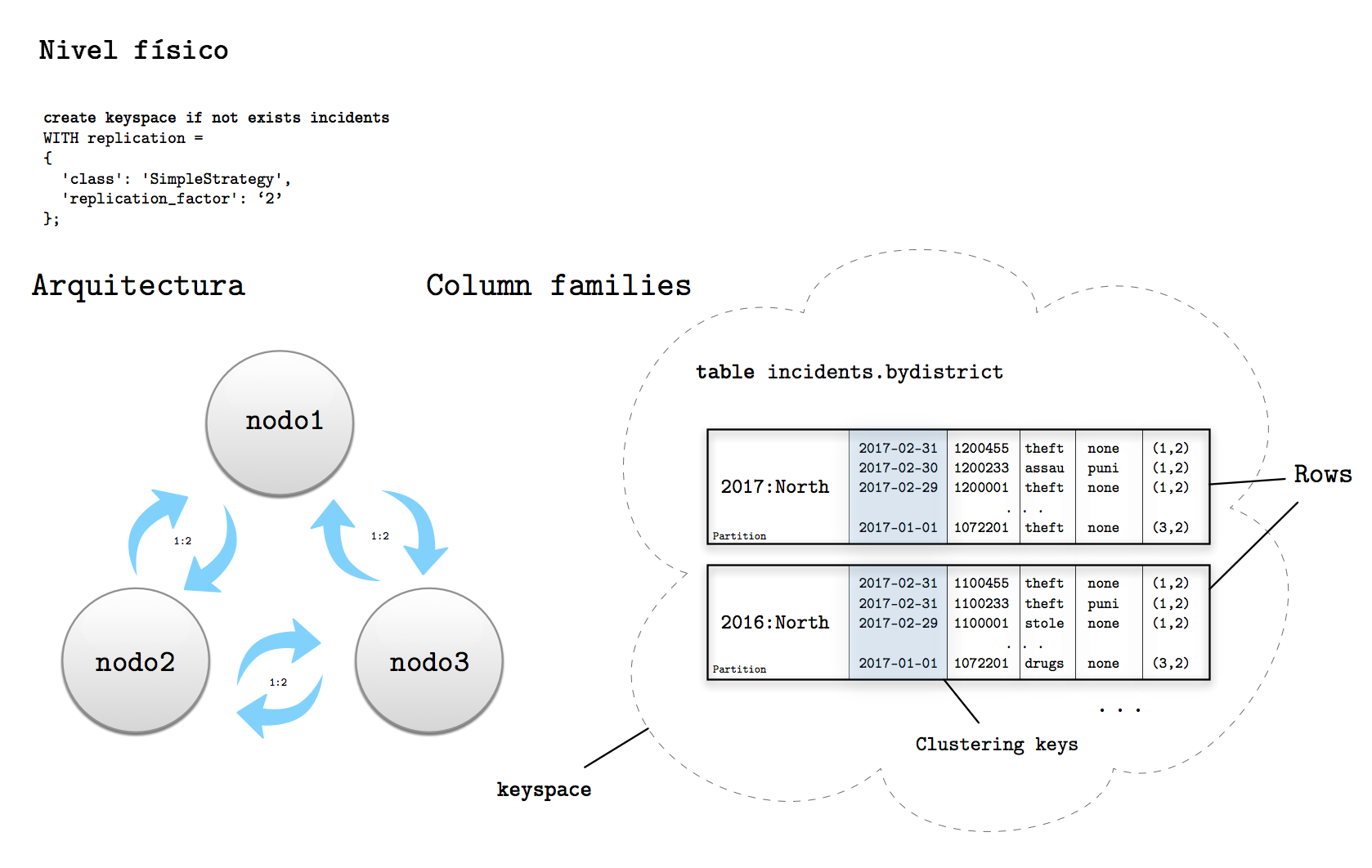
II. [Neo4j](https://github.com/orial/grupo_Spark/blob/master/Procesamiento/neo4j/readme.md#preprocesamiento-e-importaci%C3%B3n-de-datos)

III. [Mongodb](https://github.com/orial/grupo_Spark/blob/master/Procesamiento/mongodb/readme.md#preprocesamiento-e-importaci%C3%B3n-de-datos)

### I. Cassandra

Cassandra es una base de datos dependiente del caso de uso. En la mayoría de los casos, una simple instancia de MySQL o PostgreSQL haría mejor el trabajo. La idea principal es que encontremos que Cassandra pueda ofrecer facilidades con respecto a la organización de los **atributos** en nuestro modelo de datos.

**Estructura de la información**. Unas de las características principales de Cassandra es la separación entre nivel físico y lógico.

[](https://github.com/orial/grupo_Spark/blob/master/Procesamiento/docs/cassandra/nivel_fisico.png)

* **Column families** son abstracciones lógicas unitarias que conforman las tablas fisicas de CQL. Es la forma en la que disponemos la información, el esquema de organización de los campos y valores, mediante distintos tipos de columnas.
* **Particiones**. Define la estructura de compactación que engloban parte de la información de una tabla en concreto, y que las filas que la componen tienen en común el mismo criterio lógico dado por una/s clave/s de partición. Las filas que componen cada partición pueden estar ordenadas dependiendo del criterio de ordenación mediante definición de columnas como claves clustering.
* **Rows** Son las unidades de forman las column families y son instancias de las particiones. Una partición puede contener una gran cantidad de rows. Por cada partición, se pueden definir la ordenación de las filas que lo componen.

En nuestro caso, las particiones van a ser tipo **multi-row** ya que contendrán una cantidad masiva de filas ordenadas (por timestamp), y tendrán en común un determinado criterio, dado por una composite key (clave de partición compuesta): por distrito o por zona a parte del año, conformando claves de distinto tipo dependiendo del modelo de consulta que se requeiera por la aplicación:

* Por año: 2017
* Por zona/año: 2017:north
* Por tipo de delito/año: 2017:theft

La definición de las tablas necesarias para las consultas las podemos encontrar bajo el fichero:

Con respecto a las ordenación de las filas las claves de ordenación o clustering han sido fijadas sobre las columnas claves para seleccionar filas y definir cierto criterio en la realización de cualquier consulta. Generalmente la columna time ha sido seleccionada en todas las opciones y de forma DESCENDENTE, con el fin de que a nivel físico sean ordenadas de más reciente a menos reciente; pues por usabilidad no es cuestionable dicho orden.

**Arquitectura**. A **nivel de arquitectura**, solo mencionar que los datos de las tablas son alojadas como **column families** y dicha información es compartida de forma redundante entre los nodos que compongan la red Cassandra y del grado de replicación de como se configure el keyspace.

<...>

## II. Neo4j

**Neo4j** es una base de datos basada en grafos, utilizan una estructura de los lenguajes de programación conocida como *grafos*, que permite relacionar los datos a través de enlaces que facilitan el recorrido a través de ellos. Se usa para almacenar datos con relaciones complejas (por ejemplo, rutas con coordenadas GPS, relaciones sociales, ...)

### Estructura de la información

<...>

Arquitectura

<...>

## III. Mongodb

<...>

### Estructura de la información

<...>

Arquitectura

<...>

## Comparativa

En la siguiente rubrica realizamos comparaciones del tipo de arquitectura y de la capacidad de la estructuración de la información de las tres tecnologías.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | | Ventajas | | Desventajas |
| Cassandra | * **Consistencia estable y replicación de datos**. La relación entre la **estructura lógica y la física** permite que la información quede estructurada y organizada entre los nodos, con el fin de optimizar y almacenar la información entre ellos. * **Redundancia de información**. La información se encuentra organizada entre los nodos de forma redundante a su vez. * CQL Lenguaje de consulta de Cassandra es una forma bastante familiar para hacer consultas sobre Cassandra. Es un subconjunto de SQL y tiene muchas de las mismas características, haciendo que la transición de un RDBMS basado en SQL a Cassandra sea menos discordante. | | * **Sin Consultas Ad-Hoc**: La capa de almacenamiento de datos de Cassandra es básicamente un sistema de almacenamiento de clave-valor. Esto significa que debe "modelar" sus datos en torno a las consultas que desea que surjan, en lugar de en torno a la estructura de los datos en sí. Esto puede llevar a almacenar los datos varias veces de diferentes maneras para poder satisfacer los requisitos de su aplicación. * **Sin agregaciones**, aunque en esta versión si las usaremos (3.11): las versiones más nuevas de Cassandra tendrán soporte limitado para agregaciones con una sola partición. **Esto es de uso muy limitado. Debido a que Cassandra es una tienda de valores clave, hacer cosas como SUM, MIN, MAX, AVG y otras agregaciones requieren una gran cantidad de recursos si es posible**. Si hacer un análisis ad-hoc es un requisito para su aplicación, entonces Cassandra puede no ser para usted. * CQL: Es fácil para alguien que proviene de SQL confundirse acerca de qué es o no compatible. Esto significa una frustración adicional (costos de lectura) para los desarrolladores que no conocen las limitaciones de Cassandra. | |
| Mongodb | <a rellenar> | | <a rellenar> | |
| Neo4j | <a rellenar> | | <a rellenar> | |

## Conexión a la estructura de la información: conectores

Actualmente, toda estructura de almacenamiento de datos posee algún tipo de software o interfaz de usuario oficial para el mantenimiento y gestión de la información que posee.

Esta aplicación, por lo general, está construida bajo uno de los lenguajes de programación comunes ¿Cómo consigue dicho lenguaje conectarse con la base de datos e interactuar con esta? Mediante el uso de driver y conectores. Hoy en día, el término “driver”, en este contexto, quiere referirse a ambas funcionalidades de conexión e interacción, mientras que conector es el módulo software que permite interacción. Esto implica que un conector está implícitamente contenido en un driver de bases de datos.

Dichos drivers nos permitirán por tanto recuperar la información almacenada mediante el lenguaje de consulta característico de cada sistema. Tener acceso a las funciones de los conectores nos ofrece la posibilidad de diseñar funciones auxiliares basadas en estas, las cuales estructurarán y recuperarán los datos de manera que resulten útiles para el ámbito de la aplicación, dando lugar a **nuestros conectores personalizados.**

### Requerimientos comunes

Para el uso de los conectores (drivers) será necesario tener instaladas las siguientes librerias en nuestro espacio de trabajo:

* Python 3.5
* Librerías de estructuración de datos: *numpy*, *pandas, …*
* Para la representación gráfica: *matplotlib*, *seaborn, tqdm,*

### Comparativa

|  |  |  |
| --- | --- | --- |
|  | Ventajas | Desventajas |
| Cassandra | Permite balanceo de carga entre los nodos disponibles.  Pool de threads para conexiones.  Permite añadir una capa de seguridad.  Conexión única con acceso al clúster de nodos en su totalidad. | - |
| Mongodb | Capacidad de manejo de datos en formatos de documentos, con capacidad dinámica.  Capacidad de transformación de tipos de datos, sobre todo fechas.  Permite añadir una capa de seguridad.  Añade geo espacialidad para los datos relacionados con las coordenadas. | - |
| Neo4j | Permite cambios de configuración de usuario desde el conector.  Pool de conexiones que se sirven a las sesiones.  Posee comunicación mediante TLS y certificados.  Transacciones con simples o auto-commit, y explícitas, con capacidad de auto-reintento. | Driver inmutable, necesidad de crearlo de nuevo cuando hay cambios en la conexión. |

# Análisis y procesamiento de datos

## Metodología

Una vez localizada y decidida de cúal versión de la fuente de datos partiremos, se procede a su **descarga**, **preprocesamiento** o limpieza de datos, para ellao añadiremos fundamentos necesarios para realizar todo el proceso para cada tipo de representación o modelador de datos: Cassandra, Neo4j y/o Mongodb.

* **Análisis de la fuente**. Elección de la versión de la fuente de datos, siguiendo los criterios:
  + Por separación de columnas: .tsv para Cassandra, y .csv para Neo4j y Mongodb
  + Sin filtros añadidos.
* **Descarga**. Por línea de comandos o directa desde el navegador.
* **Preprocesamiento** o limpieza de datos. Mediante *bash scripting* (Cassandra) o mediante scripts de *python*(Mongodb) y *R* (Neo4j).
* **Importación al modelo de datos**. Dependerá del motor que encapsule el modelo de datos en cada caso: Neo4j, Cassandra, Y Mongodb.

## Origen y fuente de datos

Análisis de la fuente y descarga**.** Toda la información que se muestra en el mapa de incidencias, si desactivamos los filtros por defecto, se puede descargar mediante línea de comandos gracias al portal **Socrata**, lo haremos de la siguiente manera:

wget -O incidents.raw.tsv \

"https://data.sfgov.org/api/views/tmnf-yvry/rows.tsv?accessType=DOWNLOAD&api\_foundry=true"

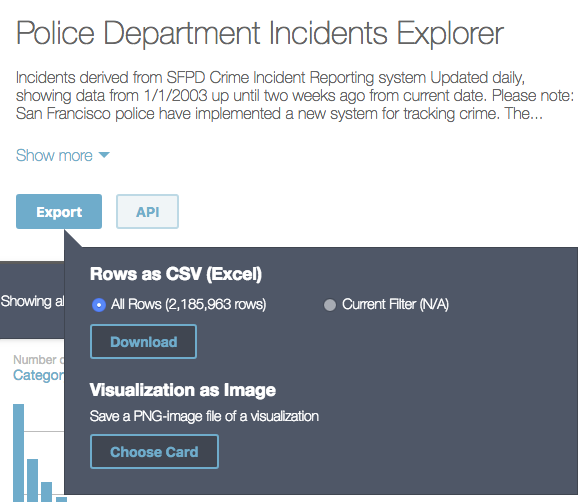
Descarga**.** La fuente de datos necesaria para realizar nuestro análisis de datos se puede descargar desde una página proporcionada por el sistema de datos del departamento policial, a través de la vista ofrecida por el portal de datos **Socrata** **Open Data**, el cual proporciona documentación para extraer reportes y las incidencias mencionadas por descargas manuales o servicios de actualizaciones a través de sus APIs. El informe sobre dicha entrada se encuentra en el portal.

Desde la página que contiene la visualización de un mapa de incidencias basada en este dataset, solamente se pueden descargar versiones del año actual. Esto ocurre desde la fecha del 3 de Marzo del 2018, bajo indicación de la última notificación, ofrecida desde la vista Socrata para dicho reporte:

[Change Notice 03/13/2018]: [Change Notice 03/13/2018]: By the end of this month, this dataset will become historical and a new one will be created starting with incident data in 2018. This one will remain here, but no longer be updated. The new one will have data coming from a new system, will not have a 2 week lag, and have updated districts among other quality improvements. We will attach a guide here with more detailed change updates as soon as we have them.

## ¿Dónde podemos encontrar la versión adecuada?

El dataset, sin filtros y con información desde el 2003 hasta la actualidad (dos semanas antes de la fecha actual), se puede descargar como documento CSV/TSV a partir de una entrada al portal proporcionado por *Socrata*:



## **Preprocesamiento e importación al modelo de datos**

La **limpieza de datos** se ha hecho de acuerdo al modelo de datos requerido para cada motor de modelado de datos: *Cassandra, Neo4j o Mongodb*; pero en general se ha seguido el mismo patrón de limpieza y han recuperado la mayoría de las columnas.

Con respecto a **la importación de datos**, para cada tecnología se ha seguido una forma de trabajar distinta en acorde al criterio seleccionado y a la discretación de ciertos atributos en acorde al modelo final.

En la fase de **diseño de la estructura** de datos, se procede con la definición del modelo de datos una vez conocidas las sentencias necesarias en la representación y visualización de la informatión, algo que puede ser característico para cada tipo de tecnológia pero a su vez deben de ceñirse a una estrategia común: **visualización de actividad criminal por intervalo de tiempo, distrito y por tipo de delito**, cuyo procedimiento lo podemos encontrar en el apartado de *"Resultados: Representación de la información"*.

A continuación se dispone de las fases y etapas realizadas por cada tecnología para llevar a cabo el procesamiento e importanción y construir finalmente un modelo de datos.

I. Cassandra

II. Mongodb

III. Neo4j

IV. Modelo programático (Cpp)

## I. Cassandra

**Instalación y configuración del entorno.** Para la carga de información y el despliegue del servidor de Cassandra se ha optado por ser realizado desde una distribución Linux. En el portal de Cassandra existen numerosas instrucciones de como realizar dicha instalación y configuración desde otras perspectivas de sistema operativo y entornos.

### Requerimientos técnicos

* Java 1.8
* Cassandra 3.11.2
* DevCenter

**Instalación**. La descarga del servidor de cassandra lo haremos a partir delcomando de descarga wget desde una distribución Linux.

$ wget http://apache.rediris.es/cassandra/3.11.2/apache-cassandra-3.11.2-bin.tar.gz

$ tar -xvf apache-cassandra-3.11.2-bin.tar.gz

* Podemos lanzar una instancia del servidor de cassandra mediante la ejecución del comando:

./apache-cassandra-3.11.2/bin/cassandra

* Para poder realizar consultas y realizar manipulaciones necesitamos lanzar el terminal CQL de la siguiente manera:

./apache-cassandra-3.11.2/bin/cqlsh

**Descarga del dataset**. El mapa de incidencias se puede descargar mediante línea de comandos de la siguiente manera.

wget -O incidents.raw.tsv \

"https://data.sfgov.org/api/views/tmnf-yvry/rows.tsv?accessType=DOWNLOAD&api\_foundry=true"

La información en CSV (con tabulador como separador) se encuentra estructurada en bruto de la siguiente manera:

IncidntNum Category Descript DayOfWeek Date Time PdDistrict Resolution Address X Y Location PdId

150060275 NON-CRIMINAL LOST PROPERTY Monday 01/19/2015 14:00 MISSION NONE 18TH ST / VALENCIA ST -122.42158168137 37.7617007179518 (37.7617007179518, -122.42158168137) 15006027571000

**Limpieza de datos.** Una vez creados los esquemas de tablas necesarias para el modelo de datos procedemos al preprocesamiento de los datos descargados desde el portal de información de incidencias para trabajar con datos limpios y en acorde al formato de los atributos que utilizaremos para dichas tablas. Se realizarán los siguientes cambios con respecto a los datos en bruto:

* Formato timestamp para definir el campo 'time', a partir de las columnas 'Date' y 'Time' ('01/19/2015' y '14:00') para obtener '2015-01-19 14:00:00'.
* Campos de agrupación relacionados con periodos de tiempo: year, month, day, hour, basados en los campos anteriores. Los cuales serán necesarios para las particiones de datos por año y/o búsqueda por horas.

Para **descargar y realizar** este limpiado de datos en el mismo lugar lo podemos realizar desde la línea de comandos de la siguiente manera:

wget -O- "https://data.sfgov.org/api/views/tmnf-yvry/rows.tsv?accessType=DOWNLOAD&api\_foundry=true" |tail -n +2 | tr '\t' ';' | sed -E 's/([0-9]+)\/([0-9]+)\/([0-9]+);([0-9]+):([0-9]+)/\3-\1-\2 \4:\5:00;\2;\3;\1;\4/g' > incidents.raw\_data.csv

o una vez descargado el dataset en 'incidents.raw.tsv' con las 2 millones de entradas, realizamos el mismo tipo de procesamiento:

$ nice cat incidents.raw.tsv |wc -l

2185964

$ nice cat incidents.raw.tsv |tail -n +2 | tr '\t' ';' \

| sed -E 's/([0-9]+)\/([0-9]+)\/([0-9]+);([0-9]+):([0-9]+)/\3-\1-\2 \4:\5:00;\2;\3;\1;\4/g' \

> incidents.dataset.csv

A partir de aquí se procede al volcado de datos a una tabla de índole general

#### Importación de datos. Para realizar el volcado e importación de datos es imprescindible qye se haya creado: el keyspace de trabajo y los esquemas y tablas físicas sobre la consola CSQL o desde el DevCenter. El comando utilizado para realizar el volcado se denomina COPY. Es un comando bastante simple de usar, poco flexible y dúctil para poder realizar transformaciones o selecciones específicas.

COPY incidents.overall(incidentId, category, description, dayoftheweek, time, day,year,month, hour, district, resolution, address,x,y,location, subid)

FROM 'dataset/incidents.dataset.sample.100.csv'

WITH DELIMITER=';' and HEADER=false and DATETIMEFORMAT='%Y-%m-%d %H:%M:%S';

Para la importación de datos a los otros esquemas se sigue un proceso de importación en cadena, de manera que entre las tablas con la información volcada son tomadas como base para realizar una exportación-importación de campos selectivos.

Tabla 1 -> Export -> solo los campos necesarios para la Tabla2 (script cql temporal) <- Import <- Tabla 2

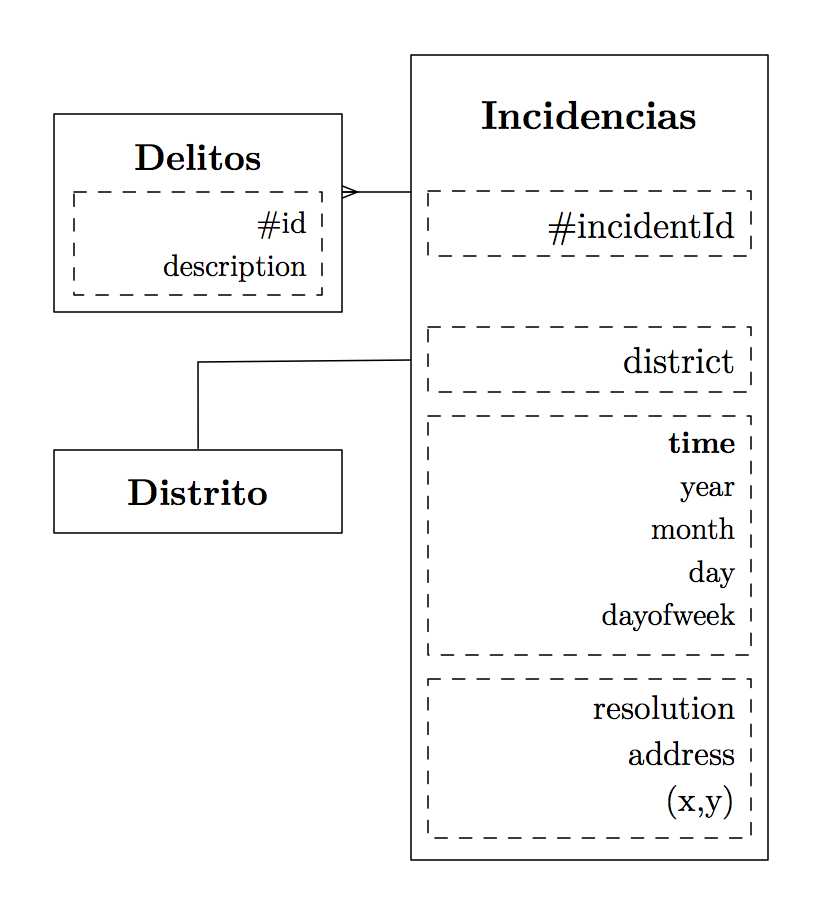
**Diseño y estructura de datos.** En Cassandra, el diseño y definición de un modelo de datos se procede una vez conocidas las metas y sentencias necesarias para la visualización final de la información a analizar. Se sugiere seguir una series de pautas para conseguir un modelado de datos idóneo para el análisis y el procesamiento masivo de datos.

Hay dos metas importantes a tener en cuenta para el modelado de datos en Cassandra:

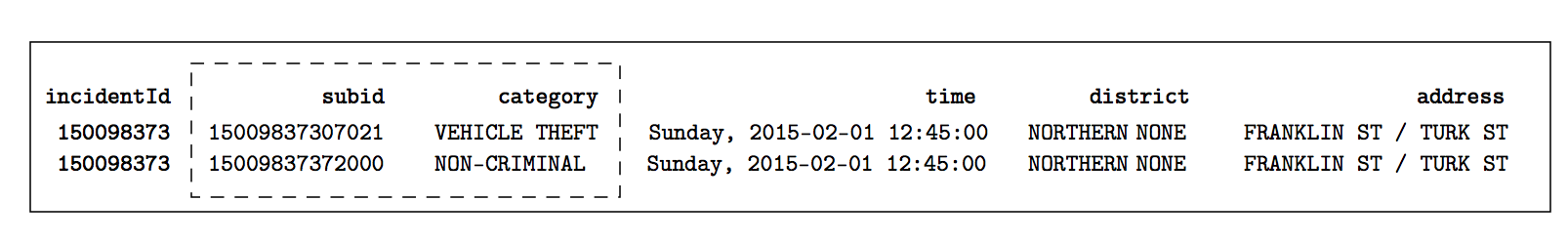
* Definir un procedimiento para repartir la información de forma equitativa y horizontal en los clusters definidos.
* Minimizar el numero de lectura de particiones.

Análisis de requisitos. Se requiere obtener y visualizar la actividad criminal para periodos de tiempo de ciertas zonasde la ciudad.

Identificar entidades y relaciones. Cada línea del documento contiene información de la relación incidencia-delito. Cada incidencia se puede desglosar en varios causas delictivas o delitos. La relación natural entre las entidades mencionadas queda representada en el siguiente diagrama:

[](https://github.com/orial/grupo_Spark/blob/master/Procesamiento/docs/modelodatos.png)

Dado que el modelo de datos se hace en acorde a las sentencias y no sobre las entidades o relaciones, no se requiere de un diseño alejado de la representación natural. En tal caso, se tendrá en cuenta la representación actual de las incidencias pero se hará para distintos escenarios (esquemas) en acorde a cada sentencia requerida.

[](https://github.com/orial/grupo_Spark/blob/master/Procesamiento/docs/modelodatosraw.png)

II. Mongodb  
  
**Instalación y configuración del entorno.** Para la carga de información y el despliegue desde mongodb se necesita de la instanación de **Mondodb** y **Compass**.

### Requerimientos técnicos

* Mongodb
* Compass

**Limpieza e importación de datos**. Descargamos csv desde la web:

* **Forma 1**: Mediante la herramienta de importación de Mongodb.

mongoimport -d datascience -c incidents --type csv \

--file Incidents.csv --headerline

* **Forma 2**: Hemos escrito un script en python2, load\_data.py que realiza la misma función.

#!/usr/bin/env python

import os

from pymongo import MongoClient

import pandas as pd

import json

import progressbar

STEP = 200

def import\_content(filepath):

# cdir = os.path.dirname(\_\_file\_\_)

# file\_res = os.path.join(cdir, filepath)

client = MongoClient()

# Creates the database for the San Francisco city incidents data

db = client['datascience']

# Creates the collection of the documents of that will represent the incidents

incid = db.incidents

# We delete any content of the incidents collection, just in case it has anything

incid.remove()

# Reads the csv file into python's dataframe type (in my case I named it incid.csv)

csv = pd.read\_csv('Incidents.csv')

bar = progressbar.ProgressBar()

for start in bar(range(0, len(csv.index), STEP)):

partial\_csv = csv.ix[start:(start + STEP), :]

# Reads the dataframe as json using orient=records,

# form info https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.to\_json.html

csv\_to\_json = json.loads(partial\_csv.to\_json(orient='records'))

# We bulk all data in

incid.insert(csv\_to\_json)

# we free some memory as csv is not needed anymore ;)

del partial\_csv

del csv\_to\_json

if \_\_name\_\_ == "\_\_main\_\_":

filepath = '~/Downloads/incid.csv'

import\_content(filepath)

#### Estructura de datos

<????>

## III. Neo4j

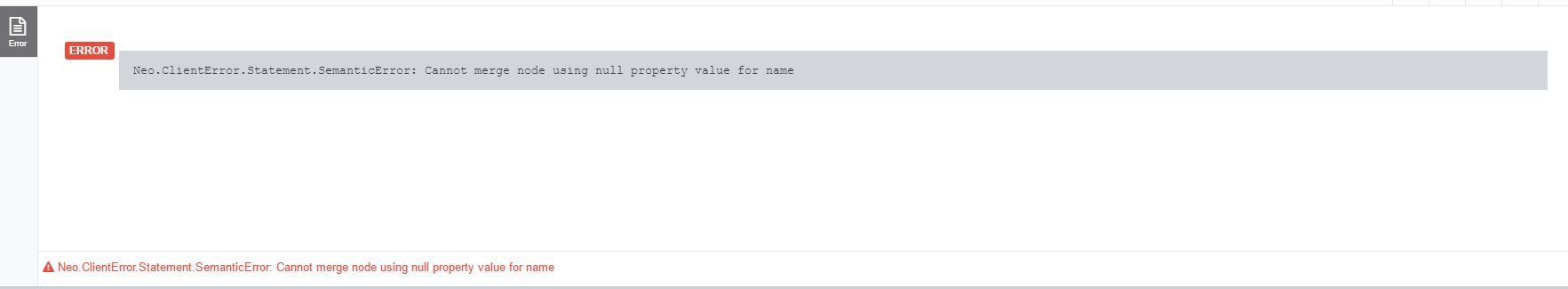
**Instalación y configuración del entorno.** Para la carga de información y el despliegue desde Neo4j, es necesario seguir los siguientes pasos:

* Download Neo4j Desktop
* Download R
* Download RStudio
* Run Neo4jDesktop
* Run RSTudio

### Requerimientos técnicos

* Neo4j Desktop/Server
* Compass

**Preparación de la información y limpieza de datos**. El preprocesamiento se realiza con ayuda del lenguaje R. Aunque en primer lugar no era necesario, tras causar cierto error en Cypherrelacionado con la ausencia de una propiedad en una operación MERGE, se procedió a buscar el motivo de este fallo en el archivo **dataset.raw.csv**descargado directamente de la fuente.



El **proceso** llevado a cabo fue el siguiente:

* En **RStudio**se importó el dataset al completo, reemplazando los valores nulos con NA. Sin embargo, al iterar sobre el dataframe, el número de filas que [**complete.cases**](https://github.com/orial/grupo_Spark/blob/master/Procesamiento/neo4j)devolvía era el mismo que el número de líneas del dataset original.

**# R**

SF\_Crime\_Heat\_Map <- read.csv("datasets/dataset.raw.csv")

View(SF\_Crime\_Heat\_Map)

missing <- SF\_Crime\_Heat\_Map[complete.cases(SF\_Crime\_Heat\_Map),]

nrow(SF\_Crime\_Heat\_Map) == nrow(missing)

**# [1] TRUE**

* Por tanto se decidió exportar estos mismos datos pero aprovechando el tipado que se había producido en la importación para ordenar el dataframe mediante el **incidentNum**. Este sería exportado a un nuevo archivo **dataset.data.csv**.

# R

SF\_data <- SF\_Crime\_Heat\_Map[order(SF\_Crime\_Heat\_Map$IncidntNum),]

View(SF\_data)

write.csv(SF\_data,"datasets/dataset.data.csv")

Este **fichero resultante** sería el utilizado para realizar el volcado al modelo de datos.

* **Nueva mejora y actualización**. Sin embargo, encontramos un problemas con el formato actual. Este consiste en que el campo Date no nos proporciona acceso individual a cada variable (Día, mes y año). Obtener estas dividiendo la columna en una sentencia cipher aumenta de forma exponencial el tiempo de carga del CSV a la base de datos, por lo tanto es recomendable realizar esta operación anteriormente y modificar el archivo existente:

# Transformamos las columas del data.frame para poder operar sobre ellas.

SF\_Crime\_Ordered\_Map\_2 <- data.frame(lapply(SF\_Crime\_Ordered\_Map, as.character), stringsAsFactors=FALSE)

# Usamos la librería stringr para mayor facilidad a la hora de separar la columna

install.packages("stringr")

library("stringr")

# Almacenamos los tres vectores resultantes

fechaSeparada <- str\_split\_fixed(SF\_Crime\_Ordered\_Map\_2$Date,'/',3)

# Añadimos al data.frame original las nuevas columnas y lo exportamos

SF\_Crime\_Ordered\_Map$Day = fechaSeparada[,2]

SF\_Crime\_Ordered\_Map$Month <- fechaSeparada[,1]

SF\_Crime\_Ordered\_Map$Year <- fechaSeparada[,3]

write.csv(SF\_Crime\_Ordered\_Map, "dataset.ordered.data.csv", row.names = FALSE)

**Importación de datos.** Una vez obtenido el fichero preformateado **dataset.ordered.data.csv**se procede a la importación desde el motor de Neo4j. Es necesario utilizar la sentencia LOAD CSV.

Pero a continuación describimos el proceso que llevamos a cabo para realizar la importación de datos y la cadena de problemas con los que nos encontramos hasta la solución final. Importamos con la sentencia LOAD CSV **básica**en su defecto.

LOAD CSV WITH HEADERS FROM 'datasets/dataset.ordered.data.csv' as line return line

Para importar los datos de CSV a la base de datos, sin embargo esto no genera nodos ni relaciones.

Para realizar la importación de datos tuvimos que seguir una serie de pasos hasta llegar con la solución final, a partir del data set **dataset.ordered.data.csv:**

**Primera solución**

Esta fue la primera sentencia **Cypher** utilizada para cargar los datos.

* El tag **WITH HEADERS** de la instrucción **LOAD CSV** nos permite referirnos a las columnas por los nombres originales situados al comienzo del csv.
* Donde **MATCH** es similar a **SELECT** y **CREATE** a **INSERT**, respectivamente, **MERGE** es una mezcla entre ambos que busca el elemento especificado, y si no lo encuentra, lo crea, lo que conlleva una búsqueda previa a la potencial inserción.
* Los elementos como nodos y relaciones se especifican con **LABELS** y **Properties**, donde los primeros actuan como tipo principal del elemento y sirven para filtrarlos más fácilmente, y los segundos almacenan datos más específicos por lo que requieren sentencias con **WHERE** para su filtrado.

LOAD CSV WITH HEADERS FROM 'datasets/dataset.ordered.data.csv' AS line

MERGE (i:INCIDENT { incidentNum:toInt(line.IncidntNum)})

SET ON CREATE i.description = line.Descript

MERGE (c:CATEGORY { name: line.Category})

MERGE (f:DATE { date: line.Date, diaSemana: line.DayOfWeek})

MERGE (r:RESOLUTION { name: line.Resolution})

MERGE (d:DISTRICT { name:line.PdDistrict})

CREATE (i)-[t:TYPE]->(c),

(i)-[p:PLACE { address:line.Address, x:line.X, y:line.Y}]->(d),

(i)-[ti:TIME {time: line.Time}]->(f),

(i)-[s:STATUS]->(r);

**Problema de esta sentencia**

* Las sentencias MERGE buscan nodos con las mismas propiedades y labels antes de insertar lo que ralentiza la carga.
* La sentencia completa intenta realizarse con la memoria RAM actual, con 2M de datos por procesar.

Esto resultaba en errores de memoria o tiempos demasiado largos para la importación (una hora y media).

Solución

* Se añadió la instrucción USING PERIODIC COMMIT la cual almacena los cambios realizados cada 1000 líneas, por defecto.
* Se añadió la instrucción ON CREATE SET que condiciona una propiedad de un nodo a su creación en Merge, mejorando la eficiencia.
* Se incluyeron índices antes de la carga para cada nodo, mejorando la velocidad de búsqueda:

CREATE INDEX ON :INCIDENT(incidentNum);

CREATE INDEX ON :CATEGORY(name);

CREATE INDEX ON :DATE(day);

CREATE INDEX ON :DATE(month);

CREATE INDEX ON :DATE(year);

CREATE INDEX ON :RESOLUTION(name);

CREATE INDEX ON :DISTRICT(name);

* + Se aumentó la memoria modificando el archivo neo4j.conf con estos valores:

dbms.memory.heap.initial\_size=2G

dbms.memory.heap.max\_size=4G

dbms.memory.pagecache.size=4G

Lo cual aumentaba la memoría máxima utilizable por **Neo4j** hasta 8Gb de RAM.

Solución final. Por tanto la sentencia final fue esta:

USING PERIODIC COMMIT 5000

LOAD CSV WITH HEADERS FROM 'datasets/dataset.ordered.data.csv' AS line

MERGE (i:INCIDENT { incidentNum:toInt(line.IncidntNum)})

SET ON CREATE i.description = line.Descript

MERGE (c:CATEGORY { name: line.Category})

MERGE (f:DATE { date: line.Date})

ON CREATE SET f.dayofweek = line.DayOfWeek

MERGE (r:RESOLUTION { name: line.Resolution})

MERGE (d:DISTRICT { name:line.PdDistrict})

CREATE (i)-[t:TYPE]->(c),

(i)-[p:PLACE { address:line.Address, x:line.X, y:line.Y}]->(d),

(i)-[ti:TIME {time: line.Time}]->(f),

(i)-[s:STATUS]->(r);

Que importaba los datos, creaba los nodos y sus relaciones en un tiempo de **3m20s**.

#### Mejora definitiva

Para cargar los nuevos datos de día, mes y año, la query tuvo que cambiar a:

USING PERIODIC COMMIT 5000

LOAD CSV WITH HEADERS FROM 'datasets/dataset.ordered.data.csv' AS line

MERGE (i:INCIDENT { incidentNum:toInt(line.IncidntNum)})

SET ON CREATE i.description=line.Descript

MERGE (c:CATEGORY { name: line.Category})

MERGE (f:DATE { day:toInteger(line.Day), month:toInteger(line.Month),year:toInteger(line.Year)})

ON CREATE SET f.dayofweek = line.DayOfWeek

MERGE (r:RESOLUTION { name: line.Resolution})

MERGE (d:DISTRICT { name:line.PdDistrict})

CREATE (i)-[t:TYPE]->(c),

(i)-[p:PLACE { address:line.Address, x:line.X, y:line.Y}]->(d),

(i)-[ti:TIME {time: line.Time}]->(f),

(i)-[s:STATUS]->(r);

La cual es ciertamente más lenta, debido a que se han añadido tipados en **:INCIDENT** y **:DATE**, además de que el **MERGE** realizado en este segundo tipo de nodo debe comparar dos variables adicionales para no causar duplicidad, aumentando la duración de la carga a **12m49s**.

¿Motivo principal?. La subida de datos a servidor es algo que sólo se debe realizar una vez, por tanto es absolutamente más eficiente que tarde más este paso, que ralentizar las búsquedas teniendo que filtrar la fecha con comparadores de cadenas de caracteres en cada una.

#### Configuración necesaria antes del import. Fue necesario tener en cuenta una serie de requisitos de configuracioón, por lo que también es importante recordar que el fichero ha de estar en la carpeta:

Application\neo4jDatabases\database-(numero)\installation-3.3.3\import  
  
Pero se puede cambiar modificando el archivo neo4j.conf en la línea:

dbms.directories.import=import

Siendo "import" el directorio por defecto.

También se puede acceder a esta configuración desde la aplicación Desktop mediante:

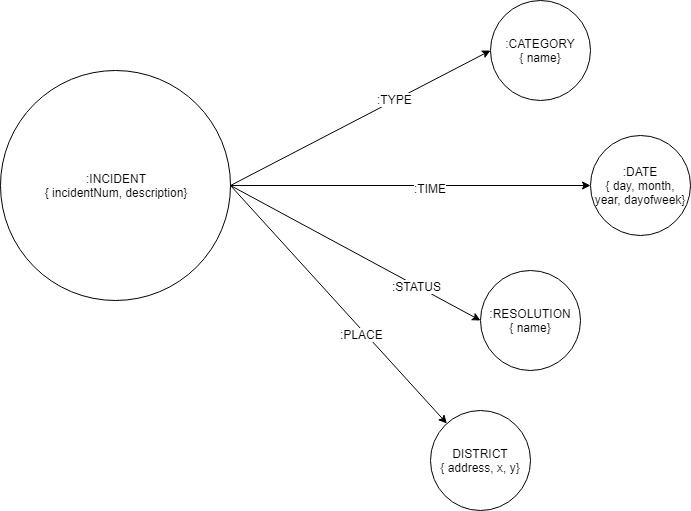
My project->(Nombre de BD)->Settings

Que provee de un editor propio.

**Diseño y estructura de datos.** Dada la estructura del archivo .CSV proporcionado, titulado con los siguientes **headers**:

IncidntNum, Category, Descript, DayOfWeek, Date, Time, PdDistrict, Resolution, Address, X, Y, Location.

Se ha decidido realizar este diseño de nodos y relaciones



Dado que, a excepción de los incidentes, un gran número de elementos estaban duplicados, se ha procedido a crear sus propios nodos y a evitar su inserción múltiple. Los datos que podían causar la aparición doble de un mismo valor, como un distrito con mismo nombre pero coordenadas X e Y diferentes a las de otro mismo distrito, se han añadido a las relaciones entre los nodos de incidentes y el resto. Esto aumenta la eficiencia y rendimiento final tanto de la inserción como de las búsquedas futuras.

Formato de fechas.Mientras que en el archivo sólo se guarda la fecha como cadena de caracteres en la columna Date, en el esquema aparecen tres campos: Day,Month y Year, los cuales serán obtenidos durante el apartado de preprocesamiento

### IV. Modelo programático (Cpp)

La aplicación processor exporta la información contenida en varios ficheros con la información detallada basada en el fichero inicial TSV recien preprocesado.

Para la instalación de la aplicación solo es necesario realizar la compilación del proyecto y de la aplicación C++.

$ g++ -g -o processor app.cpp

$ ./processor sample.tsv

**Requerimientos técnicos**

* G++
* Visual Studio Code (IDE)

**Preprocesamiento e importación de datos.** La aplicación preprocesa los datos y realiza la importación de forma automática, mediante la clase Incidents::import(filename).

$ ./processor sample.tsv

First Incident:GRAND THEFT FROM LOCKED AUTO

First occurence of Filtered Incident by dayoftheweek:GRAND THEFT FROM LOCKED AUTO,Monday

Exporting incidents count by district into filename: incidentsByDistrict.tsv ...

Incidents groups:6

Exporting incidents count by category into filename: incidentsByCategory.tsv ...

Incidents groups:1

**Estructura de datos.** La información se encuentra desglosada en estructuras basada en listas de estructura tipo **struct** ``ìncident````

struct incident

{

std::string id;

std::string category;

std::string description;

std::string dayoftheweek;

std::string time;

std::string place;

std::string resolution;

std::string address;

std::string x;

std::string y;

std::string location;

};

Mediante el método Íncidents::import(inputname) el fichero CSV es procesado y transformado en un vector de estructuras tipo incident.

# Representación de la información

Mediante las consultas y otras estructuras de representación llamadas vistas se puede obtener una visión pragmática de la información procesada. También se incluyen representaciones gráficas por lo que añadiremos todos los procedimientos llevados a cabo para realizar las presentaciones de cualquier índole. En este apartado desglosaremos este conjunto de procedimiento y pasos por tecnología, tal y como hemos ido haciendo con los apartados anteriores en acorde a las siguiente fases.

* Diseño de consultas
* Uso de conectores y construcción de consultas (python)
* Representaciones gráficas

**Consultas**. Hemos generalizado los tipos de consultas dependendiendo a las expectativas sobre la aplicación, en acorde a los siguientes usos:

* Obtener toda las incidencias para un periodo de tiempo (rango).
* Actividad criminal por zona
* Actividad criminal por tipo de delito
* Nº incidencias agrupadas por zona / *año*
* Nº incidencias agrupadas por delito / *año*

**Conectores**. Con ayuda de los conectores (drivers) y mediante Python, se pueden realizar peticiones de las consultas requeridas en la fase de diseño.

Todo el procedimiento de diseño y construcción mediante python lo encontraremos explicado de forma detallada para cada tecnología:

[I. Cassandra](https://github.com/orial/grupo_Spark/blob/master/Procesamiento/cassandra/readme.md#preprocesamiento-e-importaci%C3%B3n-de-datos)

II. [Neo4j](https://github.com/orial/grupo_Spark/blob/master/Procesamiento/neo4j/readme.md#preprocesamiento-e-importaci%C3%B3n-de-datos)

III. [Mongodb](https://github.com/orial/grupo_Spark/blob/master/Procesamiento/mongodb/readme.md#preprocesamiento-e-importaci%C3%B3n-de-datos)

IV. Processor (aplicación C++)

### I. Cassandra

### Diseño de consultas Modelado de datos Conexión con la base de datos Instalación y configuración Conectores Representación e implementación de consultas y vistas

### Diseño de consultas (Identificar sentencias). La mejor manera de particionar las lecturas es modelar los datos en acorde a las sentencias, que son dadas por los requisitos. De primera instancia hay que considerar los siguientes puntos antes de realizar la definición de las sentencias:

* Agrupación por un atributo: por distrito, por tipo de actividad criminal
* Ordenación por un atributo: por tiempo
* Filtro basado en un conjunto de condiciones.

A partir de aquí, desde otra perspectiva de alto nivel, nuestro objetivo consistirá en clasificar las requeridas incidencias y recuperar la información de la actividad criminal (cantidad/frecuencia), en periodos de tiempo, de forma ordenada, y dada ciertas condiciones definidas: por distrito o tipo de delito.

La **visualización de la actividad criminal** se consigue mediante la representación de las incidencias producidas para cada atributo: periodo de tiempo, por cada zona ó tipo de delito. Para una perspectiva de análisis un poco más precisa, se necesitaría representar la cuantificación mediante uso de proporciones o frecuencias para un determinado grupo de atributos o condiciones.

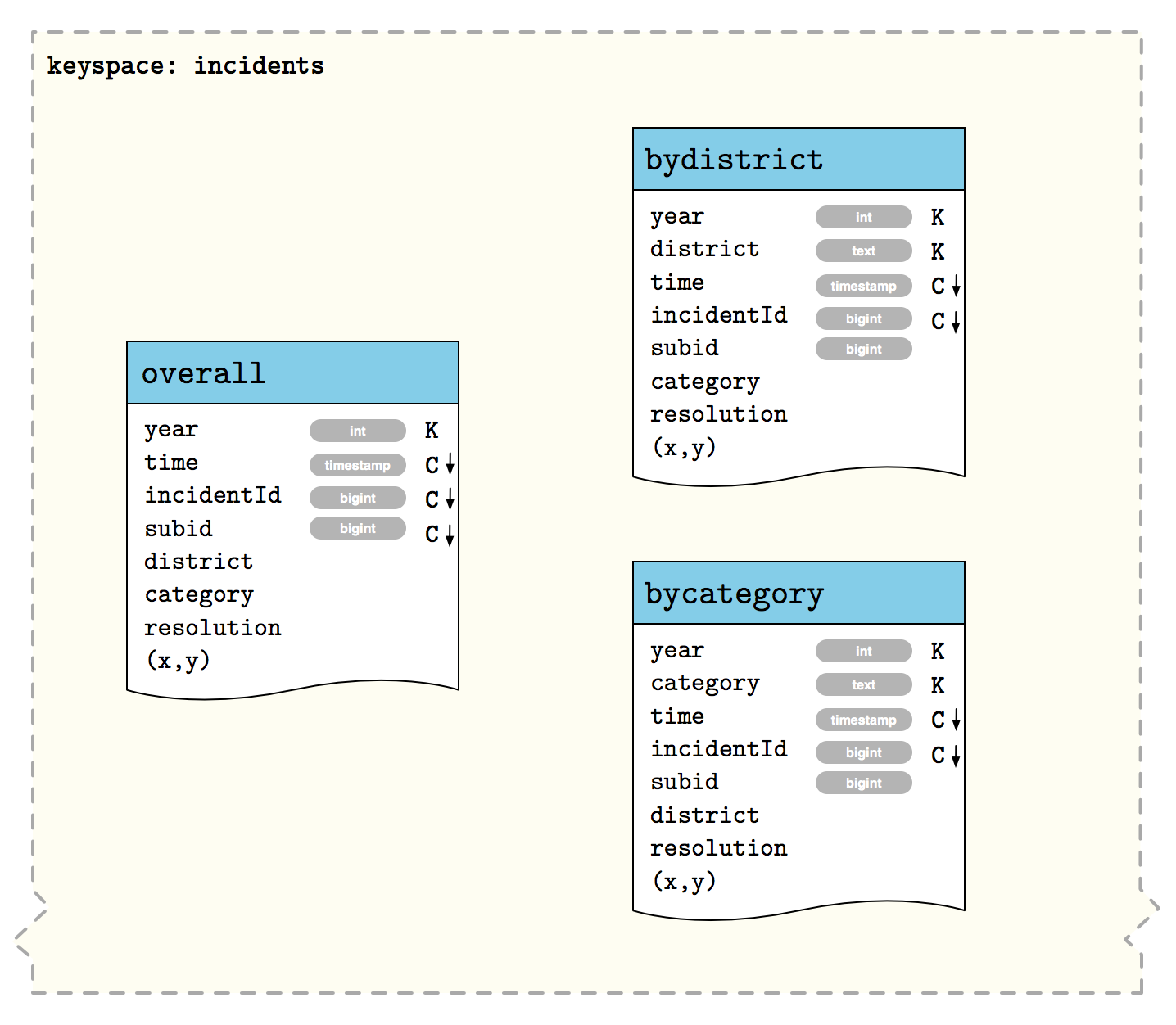
* Frecuencia criminal por periodos de tiempo: horas o día de la semana para cierto año.
* Frecuencia criminal por cada zona.
* Proporción de la incidencia de diferentes actividades delíctivas.

Especificación del esquema. La especificación del esquema vendrá determinado por la consulta o tipo de consulta que la requiera. Todos los esquemas se pueden generar obtenidos desde [schema.cql](https://github.com/orial/grupo_Spark/blob/master/Procesamiento/cassandra/cql/schema.cql)

### Modelado de datos (físico)****.**** Ahora, pasemos nuestra atención en las tablas de incidencias diseñadas. Nuestro modelo lógico contiene tres tablas desnormalizadas para admitir consultas de incidencias por año, zona, tipo de delito y fecha. A medida que trabajamos para implementar estos diseños por consultas, querremos considerar si debemos realizar la administración con la desnormalización de forma manual o usar la capacidad de extrapolar las consultas dentro de vistas en Cassandra, con ayuda de las consultas de agregación.

El diseño que se muestra para el espacio de incidencias en la siguiente figura utiliza ambos enfoques. Elegimos implementar **incidents.overall** y **incidents.bydistrict** y **incidents.bycategory** como **tablas regulares**.

El razonamiento detrás de esta elección de diseño momentáneamente, es que gracias a las claves de partición compuestas entre zona o tipo de delito con el año, podemos dividir y dimensionar mejor las consultas sobre todo cuando existe una base de información masiva.



**Conexión con la base de datos.** *El connector o driver para Cassandra se denomina cassandra.cluster*. Contiene una clase principal denominada *Cluster* que se conecta a un cluster de Cassandra estableciendo una conexión encapsulada en el objeto *Session*.

Se pueden añadir ciertas configuraciones en acorde a la arquictura definida o a la forma de conexión. Toda la información del proceso de conexión con la base de datos para la obtención de consultas lo podemos encontrar bajo el notebook: [Analisis-Cassandra.ipynb](https://github.com/orial/grupo_Spark/blob/master/Analisis/cassandra/Analisis-Cassandra.ipynb).

**Instalación y configuración**. Una vez instalado un entorno amigable a Python 3.5 necesitamos instalar las librerias necesarias:

pip install cassandra-driver

pip install django-cassandra-engine

Conexión y sesión con la base de datos. Para realizar la conexión necesitamos utilizar las clases pertenecientes de la libreria cassandra-driver.

**from** **cassandra.cluster** **import** Cluster

Una vez definida se puede permitir ealizar conexiones y consultas de forma genérica hemos implementado un conjunto de funciones auxiliares.

* La función **get\_session**encapsula la conexión sobre nuestro "keyspace" generando el objeto Session característico del conector de cassandra.

**def** get\_session(keyspace):

*"""Obtiene el conector con la sesión actual al keyspace indicado."""*

**return** (Cluster(['127.0.0.1']).connect(keyspace))

* La función **q** es una abreviatura de query y permite acondicionar la realización de consultas y parametrizar valores sobre ellas. El resultado es un objeto tipo **data frame**, por lo cual estaría totalmente adaptado al entorno y sin necesidad de ser característico para el tipo de conector o base de datos.

**def** q(session, query, \*\*kwargs):

*"""Función para encapsular las queries producidas en fmt dataframe."""*

**return** pd.DataFrame([row **for** row **in** session.execute(query.format(\*\*kwargs))])

Existen otras **funciones auxiliares**, que son necesarias para agilizar el tratamiento de las consultas sobre todo en la parte de parametrización.

* Función auxiliar para permitir un valor por defecto en el timestamp de las queries si no se introduce un valor válido. Por defecto devuelve la fecha actual como timestamp

**def** nowOrdate(date=**None**)

* Función que obtiene el operator = o in junto con los valores dependiendo si el objeto es una lista o no.

**def** eqOrInIntegers(obj=**None**)

* Función que obtiene el operator = o in junto con los valores dependiendo si el objeto es una lista o no.

**def** eqOrIn(obj=**None**)

**Representación e implementación de consultas y vistas**  
Todas las consultas se pueden encuentrar en el script CQL [cql/queries.cql](https://github.com/orial/grupo_Spark/blob/master/Procesamiento/cassandra/cql/queries.cql), pero para realizarlas es necesario crear el esquema de la estructura en la base de datos de Cassandra desde la consola a partir de los esquemas en [cql/schema.cql](https://github.com/orial/grupo_Spark/blob/master/Procesamiento/cassandra/cql/schema.cql).

1. Crear keyspace
2. Crear tablas e volcar los dataset mediante el comando COPY

Existen distintos grupos de consultas a realizar, en acorde a los atributos de tiempo, distrito y tipo de delito.

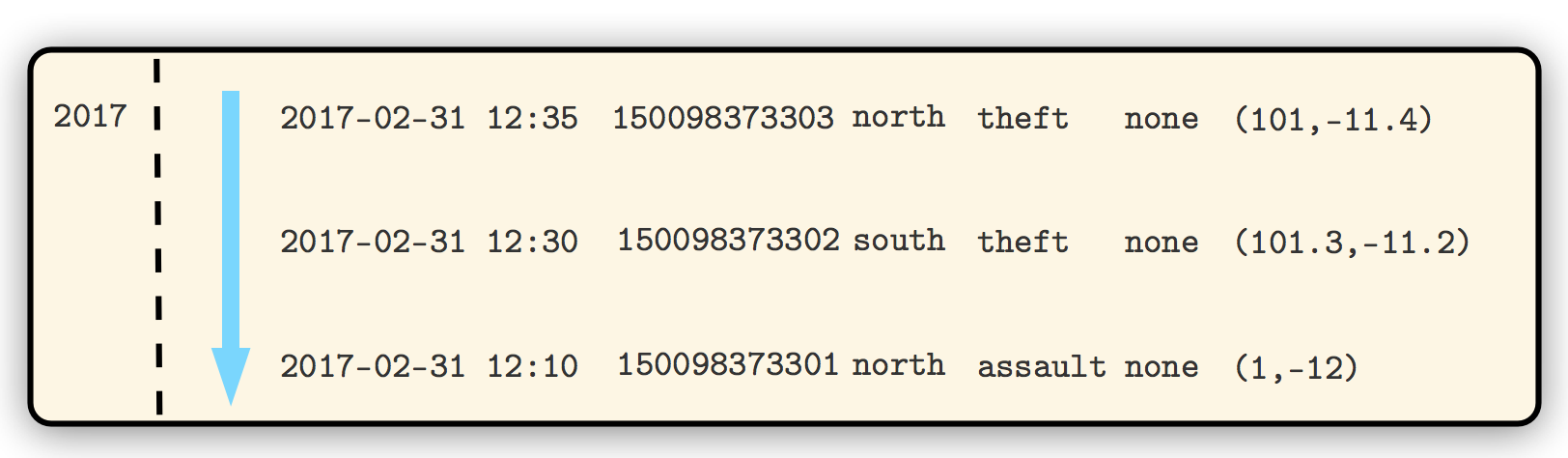
1. Periodo de tiempo

2. Por zona

### 1. Actividad criminal para un periodo de tiempo

**Diseño.** La actividad es ofrecida por la tabla: incidents.overall, con la estructura:

|  |  |
| --- | --- |
| Partition keys | year |
| Clustering keys | time,incidentid |



La **información** se encuentra particionada por una clave primaria compuesta de las claves: de partición año; pero al añadir el campo time como clave de clusterización podemos realizar una búsqueda por periodo. No se podría considerar una consulta muy eficiente ya que no se aprovecha las ventajas de particionamiento con respecto a la condición de búsqueda.

CREATE TABLE incidents.overall (

year int,

time timestamp,

...

PRIMARY KEY ((year), time, incidentId, subid)

) WITH CLUSTERING ORDER BY (time DESC, incident ASC, subId ASC);

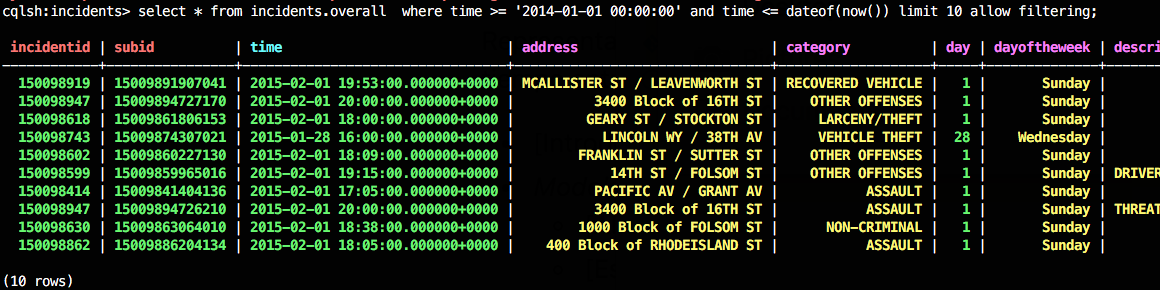
**Obtener toda las incidencias para un periodo de tiempo.**

select \* from incidents.overall

where time >= '2014-01-01 00:00:00' and time <= dateof(now())

and yearh = 2017

allow filtering;

[](https://github.com/orial/grupo_Spark/blob/master/Procesamiento/docs/cassandra/queries/query_overall_periodtime.png)

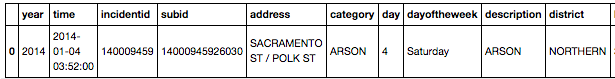
**Implementación**. Se han implementado funciones necesarias para obtener cada una de las vistas de las consultas diseñadas:

* Obtener incidencias (general)
* Obtener total de actividad criminal (getIncidents)
* Búsqueda de incidencia (getIncident)
* En un rango/periodo de tiempo (getIncidentsByDate)

**Ejemplos de llamadas a consultas por funciones mediante Python.** Gracias a las funciones mencionadas se pueden realizar consultas que previamente fueron diseñadas y ejecutadas directas a la base de datos. Mostraremos alguna de ellas:

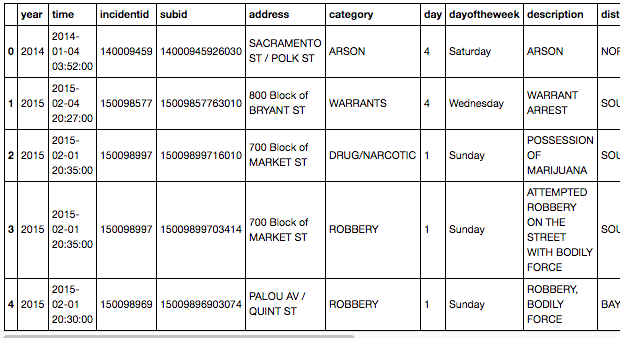
**- Obtener actividad criminal dado un identificador de incidencia**

getIncident(conn, incidentId = 140009459)



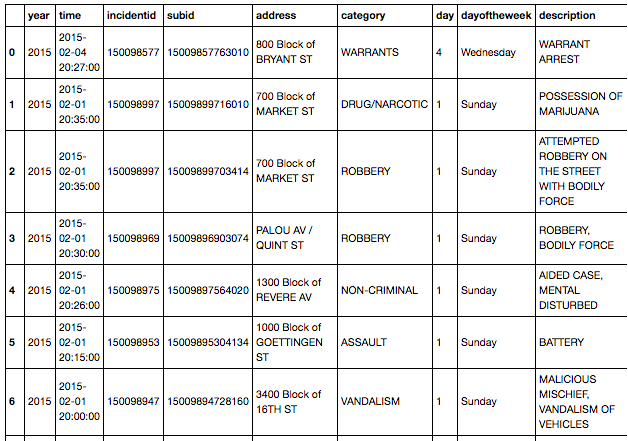
- **Obtener la actividad criminal en general (límite de 5 filas)**

getIncidents(conn, limit=5)



**- Obtener la actividad criminal desde el 2 Enero del 2015 (límite de 5 filas)**

getIncidentsByDate(conn, since="2015-01-02", limit = 5)



**2. Actividad criminal por zona**

Para esta sentencia si se realiza una partición de datos adecuada, con respecto a la zona y año: district:year. La actividad es ofrecida por la tabla: incidents.overall, con la estructura:

|  |  |
| --- | --- |
| Partition keys | year, district |
| Clustering keys | time,... |

**- Obtener información de incidencias por zonas (para un determinado año)**

select district, year, incidentid, category, time, location

from incidents.bydistrict

where year = ?



Si queremos añadir condicion de periodo de tiempo, necesitamos añadir **filtering**:

select district, year, incidentid, category, time, location

from incidents.bydistrict

where year = 2015 and time >= '2015-02-01 00:00:00' and time <= dateof(now())

allow filtering;

**Implementación**. Se han implementado funciones necesarias para obtener cada una de las vistas de las consultas diseñadas por zona:

* Obtener total de actividad criminal por distrito (getCountByDistrict)
* Obtener actividad criminal agrupado por distrito/año (getByDistrict)

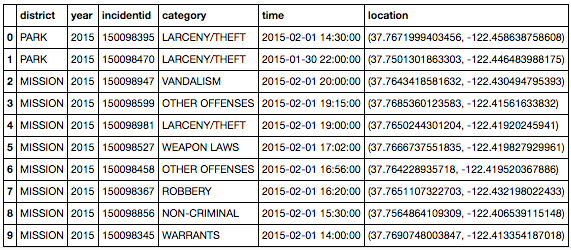
**Ejemplos de llamadas a consultas por funciones mediante Python.** Gracias a las funciones mencionadas se pueden realizar consultas que previamente fueron diseñadas y ejecutadas directas a la base de datos. Mostraremos alguna de ellas:

**- Obtener numero de incidencias por distrito (en un año determinado)**

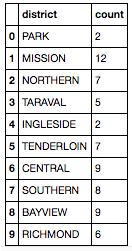
|  |  |
| --- | --- |
| getCountByDistrict(conn, limit=10) | getCountByDistrict(conn, year=[2014], limit=10) |

- **Obtener incidencias agrupadas por distrito (en un año determinado)**

getByDistrict(conn, limit=10)



getCountByDistrict(conn, limit=10)



**3. Actividad criminal por tipo de delito**

Para esta sentencia si se realiza una partición de datos adecuada, con respecto al tipo de incidencia y año: category:year.

|  |  |
| --- | --- |
| Partition keys | year, category |
| Clustering keys | time,... |

**- Obtener información de incidencias por categorias (para un determinado año)**

select category, year, incidentid, category, time, location

from incidents.bycategory

where year = ?



**- Nº incidencias agrupadas por año**

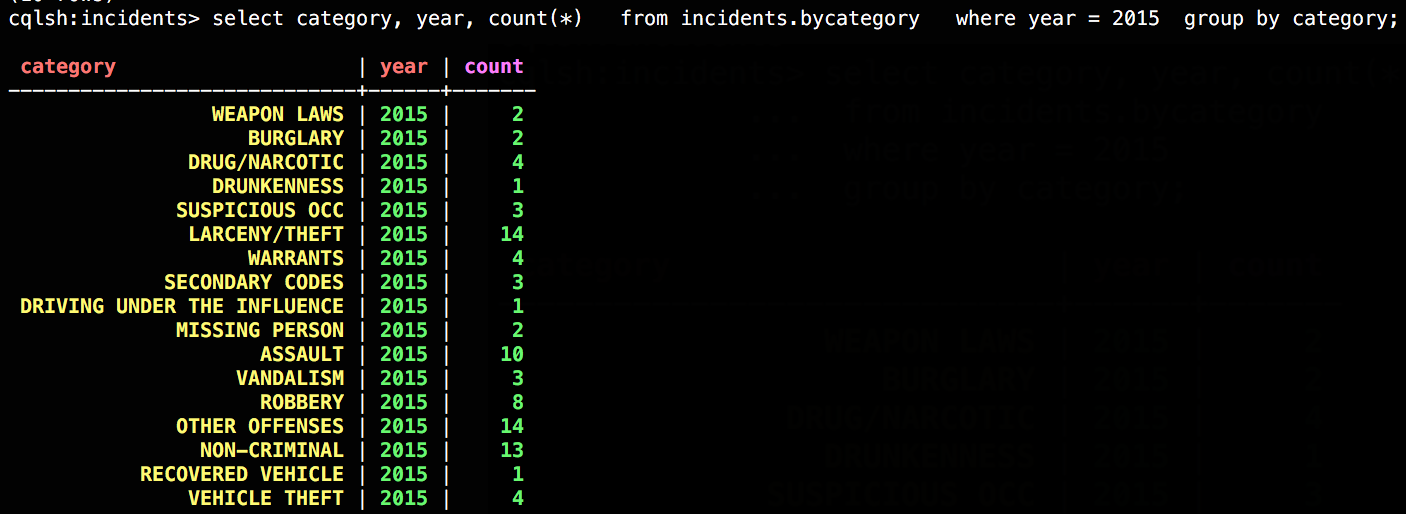
Para obtener número de incidencias producidas de cierta categoria (por un determinado año). Sin añadir filtro, más eficiente.

select category, year, count(\*)

from incidents.bycategory

where year = ?

group by category;



Nº incidencias agrupadas por **año**

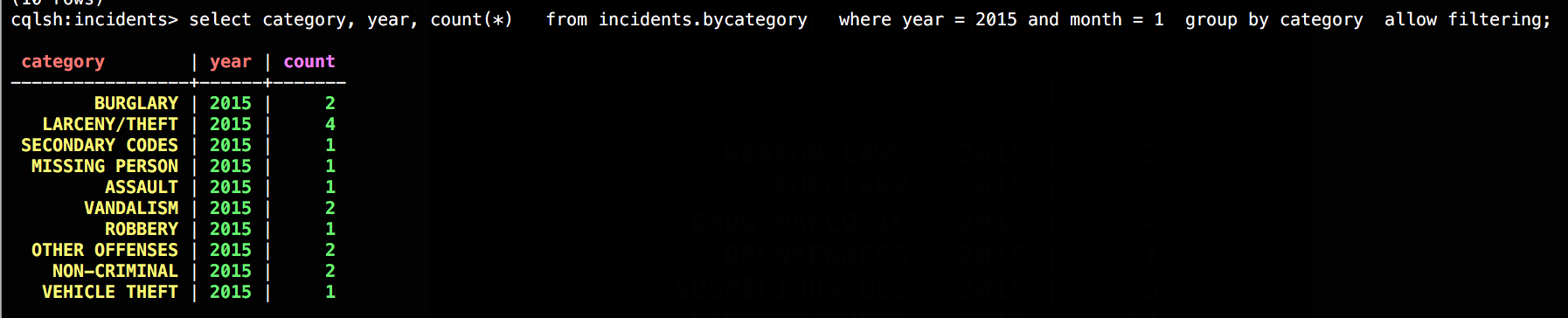
select category, year, count(\*)

from incidents.bycategory

where year = 2015 and month = 1

group by category

allow filtering;

[](https://github.com/orial/grupo_Spark/blob/master/Procesamiento/docs/cassandra/queries/query_bycategory_groupby_with_filter2.png)

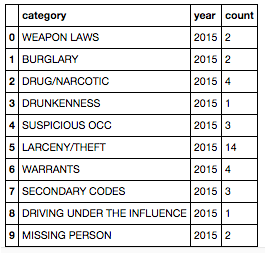
**Implementación**. Se han implementado funciones necesarias para obtener cada una de las vistas de las consultas diseñadas por tipo de delito:

* Obtener total de actividad criminal por categoria (getCountByCategory)
* Obtener actividad criminal agrupado por category/año (getRangeByCategory) permitiendo consultar un rango de fechas.

**Ejemplos de llamadas a consultas por funciones mediante Python.** Gracias a las funciones mencionadas se pueden realizar consultas que previamente fueron diseñadas y ejecutadas directas a la base de datos. Mostraremos alguna de ellas:

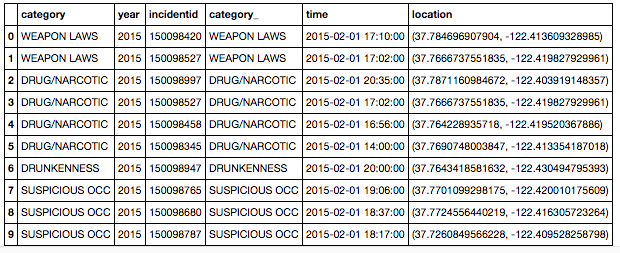
- **Obtener total de actividad criminal agrupado por cateogoria**

getCountByCategory(conn, limit = 10)

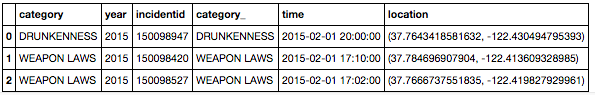


 Obtener total de actividad criminal agrupado por categoria a partir del 1 Enero de 2015

getRangeByCategory(conn, limit = 10, since= "2015-02-01")



getRangeByCategory(conn, limit = 100, since= "2015-02-01", category=["WEAPON LAWS","DRUNKENNESS"])



**II.** [**Neo4j**](https://github.com/orial/grupo_Spark/blob/master/Procesamiento/neo4j/readme.md#preprocesamiento-e-importaci%C3%B3n-de-datos)

**Diseño de consultas  
Modelado de datos  
~~Conexión con la base de datos  
 Instalación y configuración~~ Conectores  
Representación e implementación de consultas y vistas**

**Diseño de consultas**

<...>

**Modelado de datos**

<...>

**Conexión con la base de datos**. El conector **neo4j-driver** está basado en el [driver oficial de **Neo4J**](https://neo4j.com/docs/api/python-driver/current/) para **Python**. Basándose en este último, se ha creado un nuevo conector más avanzado preparado para la recolección específica de datos con el objetivo de ofrecer una mayor transparencia acerca de los objetos con los que se trabaja y mejorar la facilidad de acceso a la información consultada.

**neo4jConnector - GraphDatabase –** La clase **neo4JConnector** permite la creación de un objeto que encapsula todas las funciones que ofrece el driver de **GraphDatabase**, el cual procede de **neo4j.v1**, la librería que contiene las funciones para realizar la conexión. Se ha elegido encapsular dicho conector para abstraer los detalles sobre el procedimiento de autenticación y así simplificar el proceso. De esta forma se reduce el ámbito a la sección de real importancia: La recuperación de datos en un formato útil.

Se pueden añadir ciertas configuraciones en acorde a la arquictura definida o a la forma de conexión. Toda la información del proceso de conexión con la base de datos para la obtención de consultas lo podemos encontrar bajo el notebook: [Analisis-Neo4j](https://github.com/orial/grupo_Spark/blob/master/Analisis/neo4j/Analisis-Neo4j.ipynb).

**Instalación y configuración**. Una vez instalado un entorno amigable a Python 3.5 necesitamos instalar las librerias necesarias:

pip install neo4j-driver

**Conexión y sesión con la base de datos mediante el driver**. En Neo4j existen dos tipos de entidades sobre las que se basa el sistema de almacenamiento de datos: nodos y relaciones. Lo que significa que dentro del propio driver se han creado clases específicas para estos elementos.

**from** **neo4jConnector** **import** Neo4JConnector

Una vez definida se puede permitir ealizar conexiones y consultas de forma genérica hemos implementado un conjunto de funciones auxiliares. Gracias a la nueva clase, no es necesario especificar los parámetros, y con la URI específica de neo4j se reducen a tres argumentos por llamada de conexión.

En cada función de *Neo4JConnector* se utiliza el driver para obtener una sesión y ejecutar las llamadas a la base de datos. La existencia de este objeto nos libera de la necesidad de re-declarar funciones estáticas para las distintas consultas que se deseen hacer, y pone énfasis en qué datos se han de adquirir.

session = Neo4JConnector('bolt://neo4j.vrandkode.net:7687','test','4321')

#### 

#### Caso de uso de sesión: ejemplo de consulta

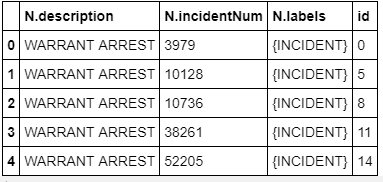
Debido al wrapper, las consultas pueden no requerir de conocer el lenguaje Cypher en el que están basadas estas últimas para Neo4j. Sin embargo, tal y como se especificó en la documentación, es posible realizar consultas especializadas también.

Para este caso, se seleccionan incidentes con una limitación en cantidad sin ningún tipo de filtro, lo que nos devuelve los primeros cinco nodos que poseen dicha etiqueta.

*# Un ejemplo de llamada: Queremos 5 nodos de incidentes*

record = session.select\_limit('INCIDENT',5)

pd.DataFrame([row **for** row **in** record])



**Representación e implementación de consultas y vistas**

Existen distintos grupos de consultas a realizar, en acorde a los atributos de tiempo, distrito y tipo de delito.

1. Periodo de tiempo (periodo, día, y por año)

2. Por zona

3. Por tipo de delito

### 1. Actividad criminal para periodos de tiempo

En este caso se obtendría el número de incidencias por día. **Count** es una función agregativa que reúne todas las entradas de un mismo tipo.

MATCH (n:INCIDENT)-[r:TIME]->(d:DATE) return d,count(n)

- Nº incidencias agrupadas por *día*

MATCH (n:INCIDENT)-->(s:DATE) return s,count(n); // Incidentes por día

- Nº incidencias agrupadas por *año*

MATCH (n:INCIDENT)-->(s:DATE) return s.year,count(n)

records = s.select\_custom('MATCH (n:INCIDENT)-->(s:DATE) return s.year as Year ,count(n) as Ocurrences ORDER BY Year')

year = list()

ocur = list()

**for** dic **in** iter(records):

year.append(dic['Year'])

ocur.append(dic['Ocurrences'])

- Actividad criminal por dia de la semana. Todales de los incidentes por dias de la semana.

MATCH (n:INCIDENT)-->(s:DATE) return s.dayofweek,s.year,count(n) order by s.dayofweek,s.year

### 2. Actividad criminal por zona

MATCH (n:INCIDENT)-[r:PLACE]->(d:DISTRICT) return distinct d,count(n)

### 3. Actividad criminal por tipo de delito

* Nº incidencias agrupadas por *año*

MATCH (c:CATEGORY)<--(n:INCIDENT)-->(s:DATE) return s.year,c.name,count(n) order by c.name

* Nº incidencias agrupadas por *día*

MATCH (c:CATEGORY)<--(n:INCIDENT)-->(s:DATE) return s,c.name,count(n) order by c.name

**Representaciones gráficas**. Un ejemplo de representación gráfica se consigue por la frecuencia de actividad criminal por tipo o año, por ejemplo, por año lo haríamos con respecto a la siguiente consulta:

**Consulta**: MATCH (n:INCIDENT)-->(s:DATE) return s.year,count(n)

**Implementación.**

records = s.select\_custom('MATCH (n:INCIDENT)-->(s:DATE) return s.year as Year ,count(n) as Ocurrences ORDER BY Year')

year = list()

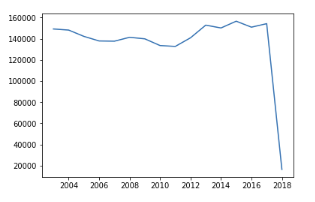
ocur = list()

**for** dic **in** iter(records):

year.append(dic['Year'])

ocur.append(dic['Ocurrences'])

plt.plot(year, ocur)



**III.** [**Mongodb**](https://github.com/orial/grupo_Spark/blob/master/Procesamiento/mongodb/readme.md#preprocesamiento-e-importaci%C3%B3n-de-datos)

**Diseño de consultas  
Modelado de datos  
~~Conexión con la base de datos  
 Instalación y configuración~~ Conectores  
Representación e implementación de consultas y vistas**

**Diseño de consultas**

<...>

**Modelado de datos**

<...>

**Conexión con la base de datos**. El paquete pymongo contiene herramientas para trabajar desde Python con Mongodb. La estructura de los datos en documentos tipo JSON con un esquema dinámico llamado BSON, lo que implica que no existe un esquema predefinido. Los elementos de los datos se denominan documentos y se guardan en colecciones.

**pymongo** – Mediante su clase principal, MongoClient, da acceso a una instancia de MongoDB, un conjunto de replicaciones o un set de “mongoses”. Dicho objeto es seguro sobre hilo y da acceso a un pool de conexiones. Soporta múltiples enlaces de mongoDB y provee de las configuraciones de cliente necesarias para establecer conexiones persistentes en segundo plano, además de poder recuperar datos en una clase de documento preestablecida.

**Instalación y configuración.**

pip install pymongo

**Configuración y creación de la sesión.** Una vez se realiza la conexión con el cliente de Mongo mediante la URL y el puerto, o el enlace específico de MongoDB, es posible acceder a la base de datos y a continuación, recuperar la información.

Las bases de datos tienen nombres únicos, por lo que la librería permite, mediante el cliente, acceder a estas como **propiedades** de un objeto. Dicha propiedad ofrece la posibilidad de adquirir los datos, que están almacenados en forma de documentos. Para ello vuelve a utilizarse una propiedad, en este caso del **objeto de base de datos**, que retorna una colección de dichos documentos.

client = MongoClient("mongodb.vrandkode.net", 27017)

db = client.datascience # Base de datos llamada datascience

collection = db.incidents # Colección de documentos

**Consultas.** Cuando accedemos a los datos de los campos de un documento en **MongoDB** podemos encontrarnos la necesidad de realizar consultas MongoDB like. Es decir, realizar consultas por cadenas similares de texto. Por ejemplo, nombres que empiecen por una letra o letras, palabras que contengan una cierta cadena de caracteres,...

Las consultas **MongoDB** like se resuelven mediante expresiones regulares. Lo que realizaremos mediante la siguinte sintaxis:

db.coleccion.find({campo:expresión\_regular});

Para los patrones de las expresiones regulares MongoDB utiliza “Perl Compatible Regular Expressions” (PCRE).

#### Caso de uso de sesión: ejemplo de consulta

Mediante la función ***aggregation***, se pueden aplicar ciertos comandos sobre la colección que retuvimos en la variable. En este caso, queremos los incidentes cuya categoría sea “robo”, e imprimimos la primera.

pipeline = [

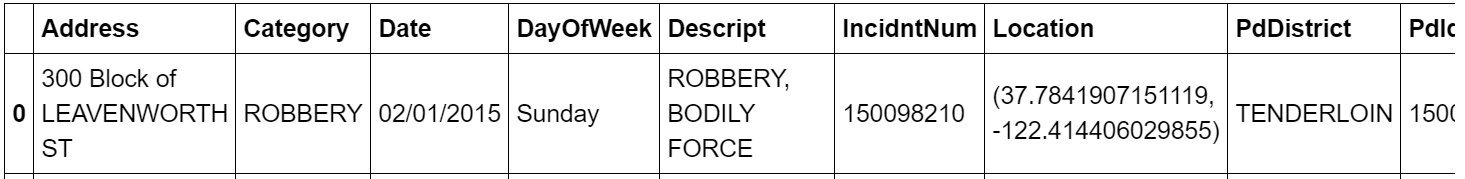
{"$match": {"Category":"ROBBERY"}},

]

aggResult = collection.aggregate(pipeline)

robbery = pd.DataFrame(list(aggResult))

robbery.head(1)



**Representación e implementación de consultas y vistas**

Existen distintos grupos de consultas a realizar, en acorde a los atributos de tiempo, distrito y tipo de delito.

1. Periodo de tiempo

2. Por zona

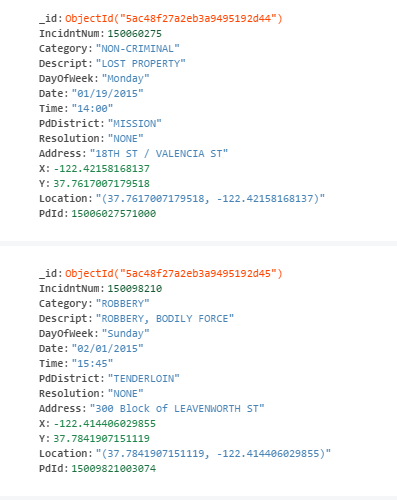
3. Por tipo de delito

### 1. Actividad criminal para un periodo de tiempo

Actividad criminal para un periodo de tiempo En este caso se obtendría el *número de incidencias* por día. **Count** es una función agregativa que reúne todas las entradas de un mismo tipo.

* Incidencias en dia Sunday

db.find({DayOfWeek:"Sunday"}).pretty()



* Número de incidencias en Sunday

db.incidents.count({DayOfWeek:"Sunday"})

290936

* Incidencias en el dia 02/01/2015

db.find({Date:"02/01/2015"}).pretty()



* Número de incidencias el 02/01/2015

db.incidents.count({Date:"02/01/2015"})

466

### 2. Actividad criminal por zona

* Incidencias en la Zona de LEAVENWORTH

db.incidents.find({Address:/LEAVENWORTH/}).pretty()

* Número de incidencias en LEAVENWORTH

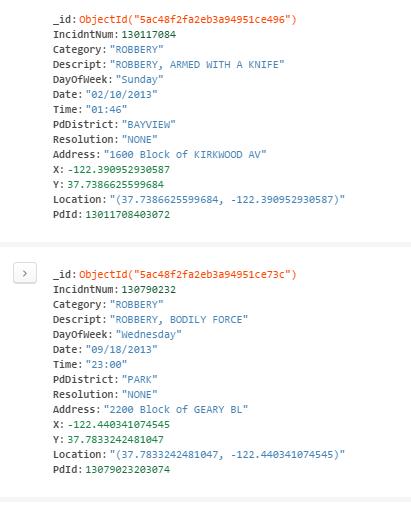
db.incidents.count({Address:/LEAVENWORTH/})

28163

### 3. Actividad criminal por tipo de delito. por tipo de delito, por año/dia (por tipo de delito)

**- Incidencias en el año 2013 para los delitos ROBBERY**

db.incidents.find({Date:/2013/,Category:"ROBBERY"}).pretty()



**Número de incidencias de la busqueda anterior**

db.incidents.count({Date:/2013/,Category:"ROBBERY"})

4196

**Distribución por tipos de delitos.** Obtenemos por cada tipo de delito una sección de datos por lo que luego podemos sacar una tabla de distribución.

db.incidents.distinct( " Category " )

['ROBBERY',

'ASSAULT'...

]

pipeline = [

{"$match": {"Category":"ROBBERY"}},

]

aggResult = collection.aggregate(pipeline)

robbery = pd.DataFrame(list(aggResult))

robbery.head()

...

pipeline = [

{"$match": {"Category":"ASSAULT"}},

]

aggResult = collection.aggregate(pipeline)

assault = pd.DataFrame(list(aggResult))

assault.head()

print("**{}** robberies (**{:.1%}**), **{}** assaults (**{:.1%}**), **{}** drugs (**{:.1%}**)" .format(

len(robbery),len(robbery)/d,

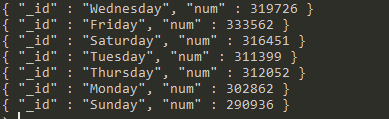
len(assault),len(assault)/d,

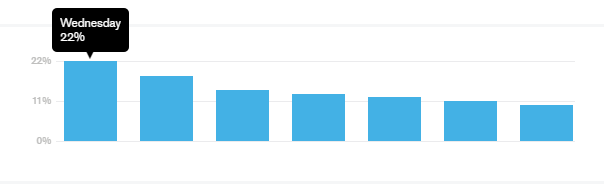
len(drug),len(drug)/d))  
  
Obtenemos como resultado la distribución por porcentajes:

55242 robberies (2.5%), 191952 assaults (8.8%), 118739 drugs (5.4%)

**Representaciones gráficas**. Se puede obtener una representación de la distribución de forma agregada por tippo de dia de la semana en formato de barras:

db.incidents.aggregate([{$group:{day:"$DayOfWeek",num:{$sum:1}}}])





**IV. Processor (aplicación C++)**

Con ayuda de los **métodos** de la clase *Incidents*: getByDistrict y getByCategory, en base al privado genérico getByAttribute; se pueden construir consultas de filtrado y de agregación de forma rápida:

/\* Filtrar y agrupar por distrito \*/

std::map<std::string, vector<incident> > getByDistrict();

/\* Filtrar y agrupar por categoria \*/

std::map<std::string, vector<incident> > getByCategory();

std::map<std::string, vector<incident> > getByAttribute(const char \* attribute);

Otros métodos de agrupación de índole genéricas son usados para realizar consultas de tipo *agregado*:

* Por zona void exportDistrictsCount(Incidents incidents, const char\* filename)
* Por distrito void exportCategoryCount(Incidents incidents, const char\* filename)

<...>

**Bibliografía**

* Python Cassandra Driver. 3.13.0 (2017). Datastax. <https://datastax.github.io/python-driver/>
* PyMongo. 3.6.1 (2018). MongoDB. <https://api.mongodb.com/python/current/index.html>
* Neo4j Bolt Driver for Python.1.6 (2018). Neo4J. <https://neo4j.com/docs/api/python-driver/current/>