Relazione del progetto per l'esame di
"Modelli e Tecniche Per i Big Data" a.a. 2022/23

# Analisi aggregata dei crimini a Chicago: Sviluppo di un'applicazione di interrogazione

Traccia:

**8-B** (framework Spark).

Studenti:

**Mattia Zicarelli** MAT. 242685

**Jacopo Alfonso Le Pera** MAT. 247521

**Silvio Emanuele Liparoti** MAT. 242663

# Analisi introduttiva

L’obiettivo del nostro progetto è quello di realizzare un’applicazione in grado di effettuare interrogazioni aggregate su un dataset contenente informazioni su un insieme di crimini registrati nella città di Chicago mediante il framework Apache Spark come previsto dalla traccia 8-B. Le analisi sono state effettuate sulla base di un dataset strutturato in formato CSV contenente i crimini commessi dal 2001 al 2021 nella città di Chicago. Di seguito sono riportati gli strumenti che l’applicazione fornisce all’utente per ricevere le statistiche sui crimini commessi nella metropoli statunitense, anche interattivamente:

1. Calcolo del numero di crimini commessi per ogni anno e relativa rappresentazione in un grafico a barre;
2. Realizzazione della classifica dei cinque crimini più frequenti in base alla tipologia di reato e relativa rappresentazione in un grafico a torta;
3. Conteggio dei crimini commessi in ogni mese di un anno specifico selezionato dall’utente e relativa rappresentazione in un grafico a linee spezzate;
4. Meccanismo di interrogazioni avanzate che permette di selezionare la tipologia di crimine e in seguito farsi restituire il computo statistico relativo a tutti gli anni o ai mesi di un anno specifico, oppure per fasce orarie;
5. Secondo meccanismo di interrogazione avanzata in cui viene scelta una Community Area della città e in seguito possono essere filtrati i risultati per mesi, anni o anche tipologie di crimine;
6. I crimini avvenuti in ogni Community Area vengono visualizzati su una mappa interattiva che associa ad ogni zona un colore la cui opacità (criminalità della zona) varia in termini di percentuale. I valori associati alla mappa vengono mostrati sul relativo grafico a barre.

# Tecnologie utilizzate

Per l’implementazione di quanto assegnatoci abbiamo deciso di usare il linguaggio di programmazione Scala. Per visualizzare i dati in un'interfaccia web è stata utilizzata un'architettura basata su servizi REST che consiste in un server e un client che lavorano insieme per creare sia il backend che il frontend dell'applicazione. Quest’ultima è formata quindi da tre componenti principali che interagiscono fra di loro:

- **Apache Spark**, una piattaforma di elaborazione distribuita per l’analisi dei dati. Come accennato in precedenza la nostra applicazione è scritta in Scala poiché è supportato da Spark e offre molteplici vantaggi tra cui: migliori performance, una più semplice integrazione con le API di Spark, stabilità, affidabilità, ma soprattutto risulta essere un linguaggio molto più conciso rispetto ad altri, quali ad esempio Java;

- Backend realizzato tramite il framework **Spring** per la gestione dei servizi web quali richieste HTTP e servizi REST esportandoli automaticamente in locale tramite la creazione di un server. Anche se questo framework nasce per lo sviluppo di applicazioni Java lo abbiamo scelto, seppur programmando in Scala, poiché ci offriva una più semplice creazione di soluzioni complesse. In particolare, abbiamo utilizzato un Controller che gestisce le richieste dell'utente richiamando le query definite nel Service che si basa sulle librerie fornite da Spark per l'analisi sul dataset.
- Frontend realizzato tramite il framework di sviluppo **Flutter** per la visualizzazione dei grafici e l'interazione con l'utente che permette di effettuare analisi statistiche come accennato nel paragrafo precedente. Flutter è stato scelto per la sua semplicità ma che garantisce al contempo un'esperienza altamente interattiva all'utente.

L'architettura descritta sopra può essere riassunta brevemente in questa sequenza di passi:

1. Una volta avviato, il server Spring, è in ascolto delle richieste da parte dell'utente sull'indirizzo IP locale sulla porta 8180.
2. Ricevute le richieste vengono ricavati i dati dal dataset mediante query statistiche scritte mediante l'utilizzo delle Spark API.
3. I risultati vengono sottoposti da Spring al frontend Flutter che si occupa della parte grafica della nostra applicazione web.

## Estrazione ed elaborazione dei dati dal dataset CSV

Il dataset a noi fornito si presenta nel formato CSV dove ogni riga indicante un crimine è composta da 22 colonne. Per ragioni di semplicità del lavoro non sono state considerate tutte le colonne del dataset in quanto alcune di esse non contenevano informazioni rilevanti ai fini della nostra analisi. Il punto di partenza del nostro lavoro è stata la creazione della classe **QueryService** che inizialmente si occupa di istanziare una sessione Spark. La SparkSession è l'entry point per l'interazione con l'API Spark che rappresenta la connessione all'ambiente Spark e fornisce metodi per creare il nostro DataFrame. La SparkSession fornisce anche funzionalità come la configurazione delle impostazioni di Spark.

```
class QueryService {
  val sparkSession: SparkSession = SparkSession.builder
    .appName("test")
    .master("local[14]")
    .getOrCreate()
```

```
val df = sparkSession.read.format("csv")
   .option("sep", ",")
   .option("inferSchema", "true")
   .option("header", "true")
   .load("C:\\Users\\utente\\Downloads\\Crimes_-_2001_to_Present.csv").cache()
```

Il file CSV che abbiamo utilizzato contiene al suo interno un dataset strutturato che presenta un header, il quale descrive le colonne al suo interno. Per questo motivo, leggendo i dati del file CSV, abbiamo deciso di creare un **DataFrame** memorizzandolo nella cache per un più facile accesso successivo.
In seguito siamo passati a definire i metodi su cui si basa la classe QueryService e che ci permettono di sottoporre le interrogazioni al nostro Dataframe. I vantaggi di utilizzare quest'ultimo piuttosto che un RDD riguardano le ottimizzazioni sottostanti effettuate da Spark SQL, come la pushdown dei filtri, la riduzione della quantità di dati trasferiti e la parziale esecuzione della query rendendo queste ultime più veloci. I DataFrame hanno uno schema definito, che descrive il tipo di dati presenti in ciascuna colonna. Ciò rende più facile l'elaborazione dei dati supportata ulteriormente dalla programmazione funzionale mediante l'utilizzo di metodi come "map", "filter", "reduce" e "aggregate".

```
val df = dataFrame.select("Date","Primary Type", "Community Area", "Year").cache()
```

Come accennato in precedenza tramite questa riga di codice abbiamo effettuato del pre-processing sul nostro Dataset scegliendo le colonne Date, Primary Type, Community Area e Year sulle quali si basa la nostra analisi.

```
def crimeAnno(dataf: DataFrame, anno: Int): DataFrame = {
   dataf.filter(s"Year==$anno").toDF()
}
def raggruppaMeseInAnno(dataf: DataFrame, anno: Int): DataFrame = {
   val specificYear = crimeAnno(dataf, anno)
   val specificMonth = specificYear.withColumn("Date", substring(col("Date"), 0, 2))
   specificMonth.groupBy(col("Date")).count().sort(col("Date").asc)  //sorted
}
def groupByAnno(dataf: DataFrame): DataFrame = {
   val conteggiati = dataf.groupBy("Year").count().sort(col("Year").asc)
   conteggiati
}
```

La prima funzione **crimeAnno(dataf: DataFrame, anno: Int): DataFrame** prende in ingresso un DataFrame e un intero rappresentante un anno, filtra le righe del DataFrame per quelle che hanno l'anno specificato e restituisce il nuovo DataFrame filtrato.
La seconda funzione **raggruppaMeseInAnno(dataf: DataFrame, anno: Int): DataFrame** prende in ingresso un DataFrame e un intero rappresentante un anno, utilizza la funzione crimeAnno per ottenere un DataFrame filtrato per quell'anno, crea una nuova colonna "Date" (rimpiazzando la precedente) che contiene solo i primi due caratteri della colonna "Date" originale (per ottenere solo il mese), quindi effettua un raggruppamento per la colonna "Date" conteggiando il numero di righe per ogni mese e restituisce il risultato ordinato per il mese. In questo modo è possibile selezionare uno specifico anno e ottenere una panoramica dei crimini commessi per ogni mese.
La terza funzione **groupByAnno(dataf: DataFrame): DataFrame** prende in ingresso un DataFrame, effettua un raggruppamento per la colonna "Year", conteggiando il numero di righe per ogni anno e restituisce il risultato ordinato per l'anno.

```
def generaClassifica(dataf: DataFrame, colName: String, n: Int): DataFrame = {
   val conteggiati = dataf.groupBy(colName).count()
   val countToList = conteggiati.select("count").map(f => f.getAs[Long](0)).collect().toList
   conteggiati.filter(col("count").geq(sortList(countToList, n))).toDF()
}
def sortList(items: List[Long], n: Int): Long = {
   items.distinct.sorted(Ordering[Long].reverse).apply(n - 1)
}
def firstFiveTypes(dataFrame: DataFrame) = {
   generaClassifica(dataFrame, "Primary Type", 5)
}
```

La funzione **generaClassifica(dataf: DataFrame, colName: String, n: Int): DataFrame** prende in input un DataFrame, un nome di colonna e un intero n e raggruppa i dati del DataFrame sulla base della colonna specificata e ne calcola il numero di occorrenze per ogni valore univoco. Questo viene fatto utilizzando il metodo groupBy sul DataFrame e il metodo count per contare le occorrenze. Effettua una mappatura sulla colonna count del DataFrame risultante, trasformandolo in una lista di Long. Filtra il DataFrame conteggiati, mantenendo solo le righe con un valore nella colonna count maggiore o uguale al n-esimo valore ordinato della lista ottenuta nel passaggio precedente. Il risultato è un nuovo DataFrame che rappresenta la classifica dei valori univoci nella colonna specificata, con i primi n valori con il più alto numero di occorrenze. Una funzione di appoggio alla precedente è **sortList(items: List[Long], n: Int): Long** che prende in input una lista di elementi Long e un intero n e restituisce l'n-esimo elemento più grande nella lista ordinata in ordine decrescente.

Infine **firstFiveTypes(dataFrame: DataFrame)** utilizza generaClassifica per ottenere la classifica dei primi 5 tipi di crimine nella colonna "Primary Type" del DataFrame.

```
def distribuzioneOrario(dataf: DataFrame): DataFrame = {
    val numMattina = crimedHours(dataf,"mattina")
    val numPomeriggio = crimedHours(dataf,"pomeriggio")
    val numSera = crimedHours(dataf,"sera")
    val numNotte = crimedHours(dataf,"notte")
    val lista: List[Long] = List(numMattina, numPomeriggio, numSera, numNotte)
    val rdd = sparkSession.sparkContext.parallelize(lista)
    val indexRdd = rdd.zipWithIndex.map { case (index,value) => (s"$index", value)
}
    indexRdd.toDF()
  }

  def crimedHours(dataf: DataFrame, fascia: String): Long = {
    val regex = getFasciaOraria(fascia)
    val result = dataf.select($"Date", regexp_extract($"Date", regex,
0).alias("match"))
      .filter(col("match").startsWith("0").or(col("match").startsWith("1"))).count()
    result
  }

  def getFasciaOraria(fascia: String): String = {
    fascia match {
      case "mattina" =>
        "07:\\d{2}(:\\d{2})?       AM|08:\\d{2}(:\\d{2})?       AM|09:\\d{2}(:\\d{2})?
AM|10:\\d{2}(:\\d{2})? AM|11:\\d{2}(:\\d{2})? AM|12:\\d{2}(:\\d{2})? AM"
      case "pomeriggio" =>
        "01:\\d{2}(:\\d{2})?       PM|02:\\d{2}(:\\d{2})?       PM|03:\\d{2}(:\\d{2})?
PM|04:\\d{2}(:\\d{2})? PM|05:\\d{2}(:\\d{2})? PM|06:\\d{2}(:\\d{2})? PM"
      case "sera" =>
        "07:\\d{2}(:\\d{2})?     PM     |08:\\d{2}(:\\d{2})?     PM|09:\\d{2}(:\\d{2})?
PM|10:\\d{2}(:\\d{2})? PM|11:\\d{2}(:\\d{2})? PM|12:\\d{2}(:\\d{2})? PM"
      case "notte" =>
        "01:\\d{2}(:\\d{2})?       AM|02:\\d{2}(:\\d{2})?       AM|03:\\d{2}(:\\d{2})?
AM|04:\\d{2}(:\\d{2})? AM|05:\\d{2}(:\\d{2})? AM|06:\\d{2}(:\\d{2})? AM"
    }
  }
```

Questo codice  analizza i crimini commessi in una città e calcola la distribuzione oraria dei crimini. Ci sono 3 funzioni principali:

1. **distribuzioneOrario(dataf:DataFrame): DataFrame** questa funzione chiama 4 volte la funzione crimedHours per ottenere il numero di crimini commessi in diverse fasce orarie (mattina, pomeriggio, sera e notte), crea una lista di questi numeri e li converte in un DataFrame.
2. **crimedHours(dataf: DataFrame, fascia: String): Long** questa funzione prende in input un DataFrame e una stringa che indica la fascia oraria da considerare. Utilizza una espressione regolare per filtrare le righe del DataFrame che corrispondono alla fascia oraria specificata e restituisce il numero di righe filtrate.
3. **getFasciaOraria(fascia: String): String** questa funzione prende in input una stringa che indica la fascia oraria da considerare e restituisce una espressione regolare che è utilizzata per filtrare le righe del DataFrame che corrispondono alla fascia oraria specificata.

## Gestione delle richieste e trasmissione dei dati

Arrivati a questo punto, una volta descritta la classe QueryService utilizzata per elaborare i dati, passiamo all'analisi della classe **CrimesController** che si occupa della gestione dei servizi web quali REST e HTTP richiesti dall'utente, richiamando le query definite nel Service.

```
@RestController
@RequestMapping()
class CrimesController {
      val queryService = new QueryService()
```

CrimesController è un controller REST scritto in Scala che utilizza la libreria Apache Spark per lavorare sui dati. Il controller utilizza l'annotazione @RestController per identificare la classe come controller REST e l'annotazione @RequestMapping per definire l'URL di base per le richieste al controller.

```
  @GetMapping(Array("/types"))
  def getPrimaryTypes: String = {
    val classificaTipi = queryService.firstFiveTypes(queryService.df)
    tojsonString(classificaTipi)
  }
```

Questa funzione viene eseguita quando viene effettuata una richiesta HTTP GET all'URL "/types". Questo metodo chiama il metodo "firstFiveTypes" del QueryService

per ottenere la classifica dei cinque tipi di crimini più comuni, trasforma il risultato in una stringa JSON e la restituisce come risposta alla richiesta. Ciò è possibile grazie alla funzione **tojsonString(dataFrame: DataFrame) :String**.

```
def tojsonString(dataFrame: DataFrame) :String = {

    val jsonresult = dataFrame.toJSON.collect().mkString(",")

    "[" + jsonresult + "]"

  }
```

Questo codice definisce una funzione Scala chiamata "tojsonString" che prende come input un oggetto DataFrame di Apache Spark e restituisce una stringa in formato JSON.La funzione esegue le seguenti azioni. Utilizza il metodo "toJSON" sull'oggetto DataFrame per convertire i dati in una rappresentazione JSON.Utilizza il metodo "collect" sull'oggetto JSON per recuperare tutti i dati come una serie di stringhe.Utilizza il metodo "mkString" sulla lista di stringhe per concatenare tutte le stringhe in un'unica stringa separata da virgole.Infine, la funzione restituisce una stringa in formato JSON che inizia con "[" e termina con "]" (che rappresentano i delimitatori di un array JSON).

Sulla falsa riga della funzione **getPrimaryTypes: String** vengono riportate le funzioni implementate nel controller:

```
@GetMapping(Array("/years"))

 def getByYear(): String = {

   val groupedByYear = queryService.groupByAnno(queryService.df)

  tojsonString(groupedByYear)

 }

@GetMapping(Array("/months"))

 def getMonthsByYear(@RequestParam(value = "year", defaultValue = "2001") year: Int):
String = {

   val groupedByMonth = queryService.raggruppaMeseInAnno(queryService.df, year)

  tojsonString(groupedByMonth) }
```

**getByYear(): String** viene eseguita quando viene effettuata una richiesta HTTP GET all'URL "/years". Chiama il metodo "groupByAnno" del QueryService per ottenere i dati raggruppati per anno. Utilizza la funzione "tojsonString" per convertire i dati ottenuti in una stringa in formato JSON. Restituisce la stringa in formato JSON come risposta alla richiesta HTTP. **getMonthsByYear(@RequestParam(value = "year", defaultValue = "2001") year: Int): String** prende un parametro opzionale "year" tramite @RequestParam e utilizza un valore predefinito di "2001" se non viene fornito un parametro. La funzione chiama due metodi su un oggetto "queryService": "raggruppaMeseInAnno" e "tojsonString".

Seguendo la struttura delle precedenti funzioni sono state implementate tutte le altre delle quali, per brevità, riportiamo solo l'intestazione.

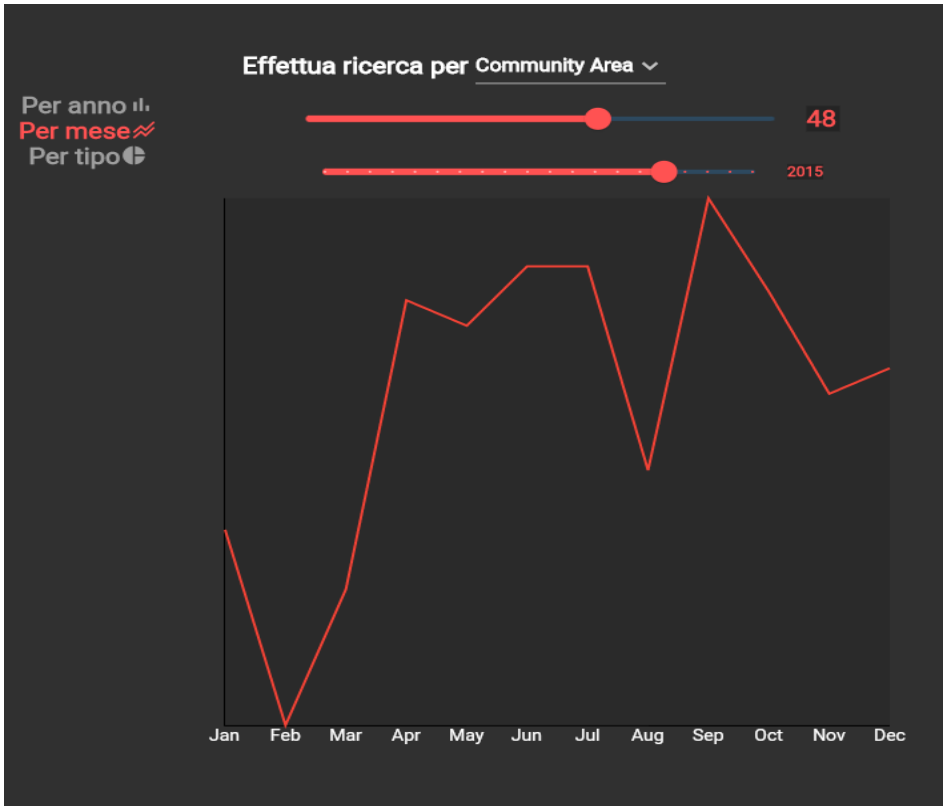
```
 @GetMapping(Array("/yearsByType"))

 def getYearsByType(@RequestParam(value = "tipo") tipo: String): String = {…}

@GetMapping(Array("/yearsByArea"))

 def getYearsByArea(@RequestParam(value = "area") area: String): String = {…}

@GetMapping(Array("/getMonthByType"))

 def getMonthsByYearAndTipo(@RequestParam(value = "year") year: Int, @RequestParam(
value = "tipo") tipo: String): String = {…}

@GetMapping(Array("/getMonthByArea"))

 def getMonthsByYearAndArea(@RequestParam(value = "year") year: Int,

 @RequestParam( value = "area") area: String): String = {…}

 @GetMapping(Array("/getTypesByArea"))

 def getTypesByArea(@RequestParam( value = "area") area: String): String = {…}

 @GetMapping(Array("/getOrariByType"))

 def getOrari(@RequestParam(value="tipo") tipo: String): String = {…}

 @GetMapping(Array("/getAreas"))

 def getAreas():String={…}
```

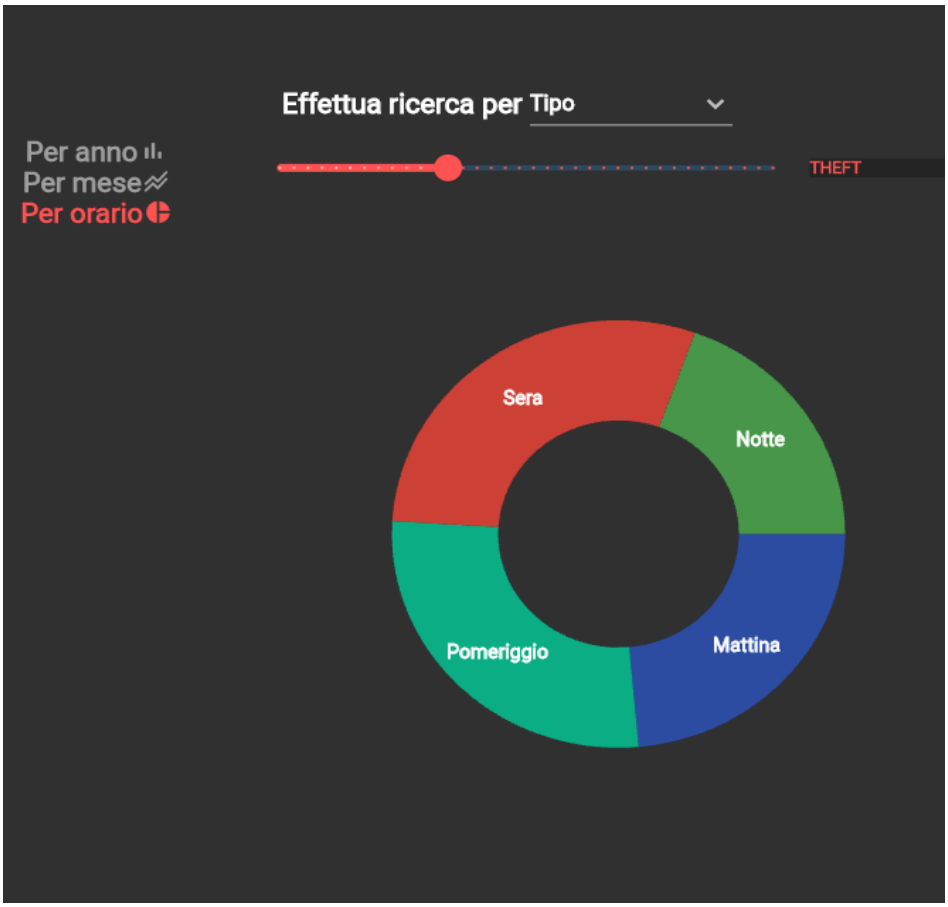
# Visualizzazione tramite Web Application

Per quanto riguarda la visualizzazione dei risultati di analisi statistica abbiamo implementato un frontend attraverso il framework Flutter, seguendo il design pattern di sviluppo MVC (Model-View-Controller). L'utilizzo di questo framework ci ha permesso di definire un REST manager che si occupa di gestire le chiamate HTTP effettuate dall'utente, inoltrandole al controller del backend. Dopo aver ricevuto i risultati, l'utente può visualizzarli attraverso i vari widget che abbiamo progettato. In particolare, sono stati implementati, come già descritto all'inizio della nostra trattazione grafici a barre, a torta e a linee spezzate.

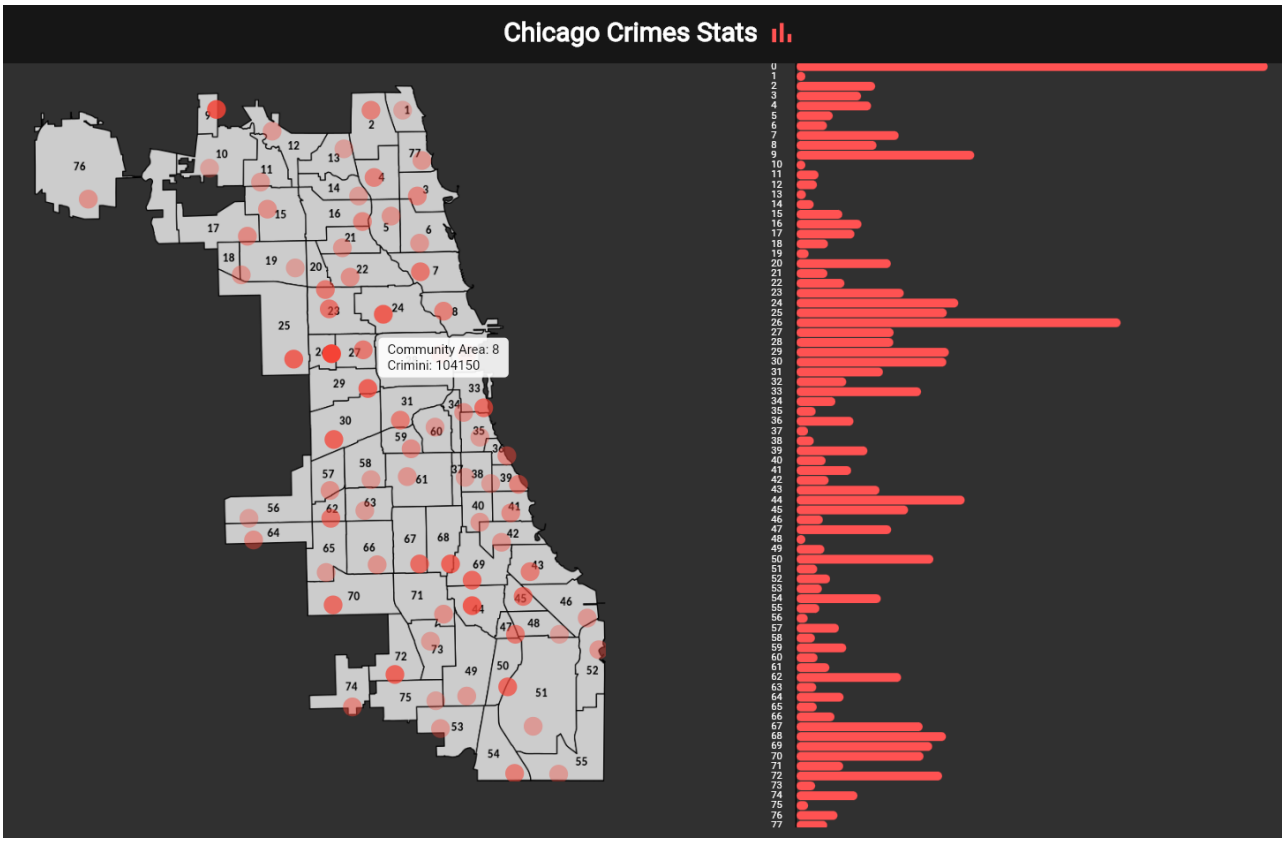
La nostra applicazione web si compone fondamentalmente di tre sezioni in un'unica home page. In alto abbiamo il nome del nostro sito web e tre grafici generati attraverso una chiamata automatica al dataset completo.

Successivamente vi sono dei widget che consentono all'utente di interagire ed effettuare delle interrogazioni avanzate selezionando vari filtri.



Infine, è presente una mappa interattiva che permette all'utente di visualizzare i crimini associati ad ogni community area. È possibile attraversare con il puntatore del mouse ogni zona per visualizzare i suddetti crimini mostrati anche su di un grafico a barre correlato alla mappa.




# Conclusioni

L'analisi statistica dei dati sulla criminalità può offrire soluzioni per affrontare questo problema. Utilizzando i trend e le informazioni sulla distribuzione e la natura dei crimini, è possibile identificare le aree e i periodi a maggior rischio e prendere misure preventive. Inoltre, l'analisi statistica può anche supportare lo sviluppo di politiche e programmi per prevenire la criminalità, come l'implementazione di programmi di prevenzione della violenza, la riforma del sistema giudiziario e la collaborazione con le comunità per rafforzare la sicurezza. L'analisi statistica può anche essere utilizzata per monitorare l'efficacia delle soluzioni adottate e per adattare e migliorare le strategie in base alle esigenze emergenti. In sintesi, l'analisi statistica della criminalità è uno strumento potente per prevenire la criminalità e garantire la sicurezza della comunità.

In conclusione, la realizzazione di questa applicazione ci ha consentito di apprendere come operare sui dataset strutturati, effettuare aggregazioni ed analisi, manipolare grandi quantità di dati e renderli disponibili alla visualizzazione da parte dell'utente tramite la piattaforma di analisi distribuita Apache Spark.