Advanced Data Management Project: Crime Monitoring and Analysis in Los Angeles.

Miggiano Davide 4840761
Morando Andrea 4604844

Contents

Video Presentation Link

Proposed Domain and Application

The Application we propose is a Crime and Demographic relationship for Los Angeles Police Department (LAPD). The system provides:

- Social analysis of crime, including demographic information to help law enforcement understand crime behaviors.
- Connection between Crime type, weapon used and victims
- Impact of cops distribution across the street with certain type of victims and crime

Application details

Main characteristics:

- **Read/Write Nature**: The application is read-intensive, with frequent relational queries from crime type, victims and street/district. The writes volume is relatively low, around 650 crimes each day.
- **System Requirements**:
  - **Consistency**: Strong consistency is required for certain operations (e.g., ensuring accurate crime count in each location), but eventual consistency might be acceptable for non-critical operations like weekly report.
  - **High Availability**: High availability is critical to ensure that the system remains accessible for any kind of emergency.

Conceptual Schema

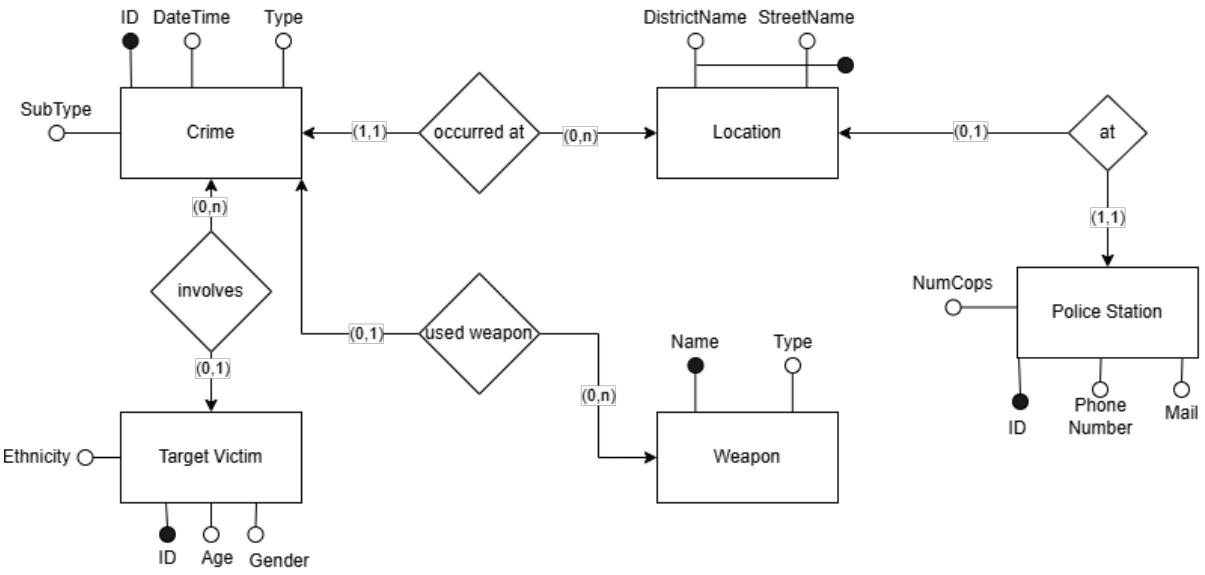

Figure 1: ER Schema

Workload

1. Find all crimes and their type that occurred in a given district and in a given time interval;
2. Identify all victims given a crime type;
3. Count all crimes involving a certain type of weapon;
4. Find the crimes and victims of a given district;
5. Given an ethnicity find all streets in which that ethnicity has experienced crimes, sorting them by the number of crimes.
6. Find the number of cops and all districts where the Crime (Type) "Assault" was done with a weapon of type "FIREARM".

Aggregate-oriented Design Methodology

Q1: (Crime, [Crime(DateTime)_!, Location(DistrictName)_OA], [Crime(ID, Type)_!])
Q2: (Crime, [Crime(Type) _!], [TargetVictim(Age, Gender, Ethnicity) _I])
Q3: (Weapon, [Weapon(Type) _!], [Crime(ID) _UW])
Q4: (Location, [Location(DistrictName) _!], [Crime _OA, TargetVictim _IOA])
Q5: (TargetVictim, [TargetVictim(Ethnicity)_!], [Location(StreetName)_OAI, Crime(ID)_I])
Q6: (Crime, [Crime(Type)_!, Weapon(Type)_UW], [Location(DistrictName)_OA, PoliceStation(NumCops)_AOA])

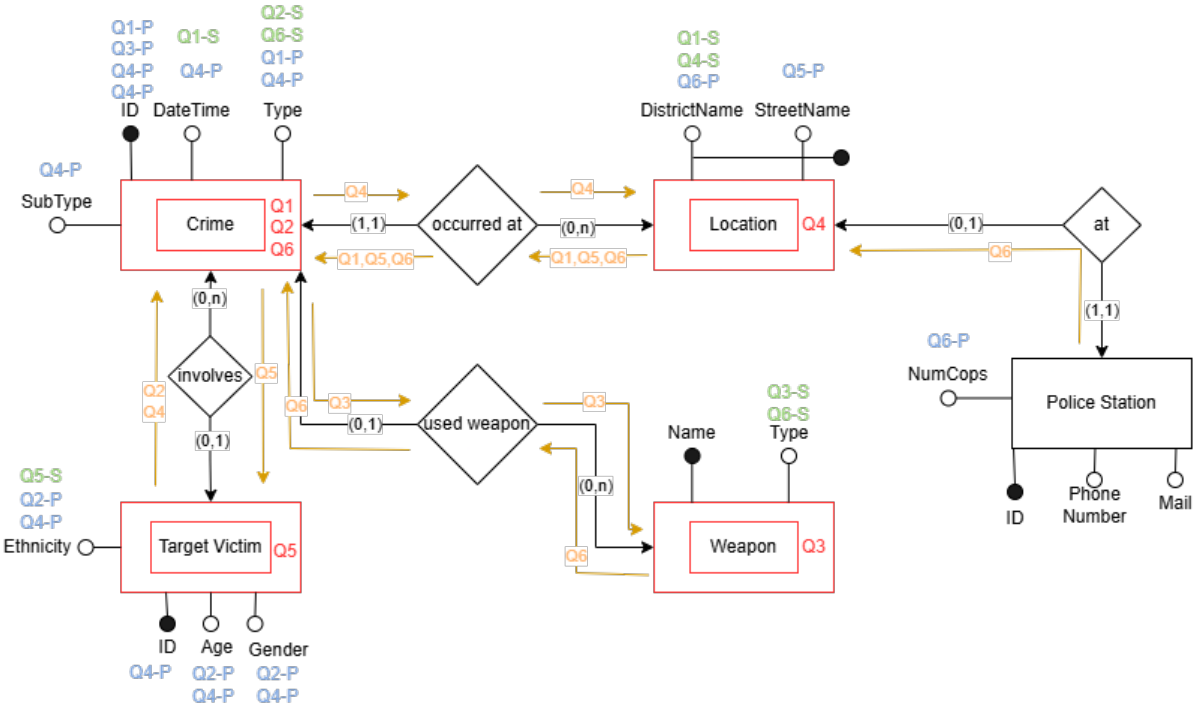

Figure 2: Annotated ER Schema

- ***crime***: {Id, DateTime, Type, DistrictName, NumCops, WeaponType, Victim: [{Ethnicity, Age, Gender}]}
- ***weapon*** : {Name, Type, CrimeIds : [{Id}]}
- ***location*** : {DistrictName, StreetName, Crimes : [{Id, DateTime, Type, Subtype}], TargetVictim : [{Id, Age, Gender, Ethnicity}]}
- ***target victim*** : {Id, Ethnicity, CrimeId, StreetName}

Design in MongoDB

All the obtained aggregates do not need any changes for MongoDB, since any JSON notation can be used. For all the documents an _id field is automatically assigned and indexed.

	Crime	Weapon	Location	TargetVictim
Q1	DateTime		DistrictName	
Q2	Type			
Q3		Type		
Q4			DistrictName	
Q5				Ethnicity
Q6	Type	Type		

Table 1: Selection Attributes table

Shard Keys and Indexes:

Crime: Creation of a shard key on {CrimeType}, two non unique indexes on {CrimeType} and {DateTime} and an unique index on {CrimeType, Id}.

```
-- Create non-unique index on CrimeType and a non-unique index on DateTime
db.crime.createIndex({ Type: 1 });
db.crime.createIndex({ DateTime: 1 });

-- Create unique index on CrimeType and Id
db.crime.createIndex({ Type: 1, Id: 1 }, { unique: true });

-- Shard the crime collection on CrimeType
sh.shardCollection("db.crime", { Type: 1 });
```

Weapon: Creation of a shard key on {Type}, one non unique index on {Type} and a unique index on {Type, Name}

```
-- Create non-unique index on Type
db.weapon.createIndex({ Type: 1 });

--Create an unique index on WeaponType and Name
db.weapon.createIndex({ Type: 1, Name: 1 }, { unique: true });

-- Shard the weapon collection on Type
sh.shardCollection("db.weapon", { Type: 1 });
```

Location: Creation of a shard key on {DistrictName} and an unique index on {DistrictName}

```
-- Create unique index on DistrictName
db.location.createIndex({ DistrictName: 1 }, { unique: true });

-- Shard the location collection on DistrictName
sh.shardCollection("db.location", { DistrictName: 1 });
```

TargetVictim: Creation of a shard {Ethnicity}, a non unique index on {Ethnicity} and an unique index on {Ethnicity, Id}

```
-- Create non-unique index on Ethnicity and an unique index on Ethnicity and Id
db.targetvictim.createIndex({ Ethnicity: 1 });
db.targetvictim.createIndex({ Ethnicity: 1, Id: 1 }, { unique: true });

-- Shard the targetvictim collection on Ethnicity
sh.shardCollection("db.targetvictim", { Ethnicity: 1 });
```

Queries:

1. Find all crimes and their type that occurred in a given district and in a given time interval;

```
db.crime.find({
  DistrictName: "GIVEN_DISTRICT",
  DateTime: { $gte: ISODate("START_DATE"), $lte: ISODate("END_DATE") }
}, {CrimeType: 1, DateTime: 1});
```

2. Identify all victims given a crime type;

```
db.crime.aggregate([
  { $match: { CrimeType: "GIVEN_CRIME_TYPE" } },
  { $unwind: "$Victim" },
  { $project: { _id: 0, "Victim.Ethnicity": 1, "Victim.Age": 1,
      "Victim.Gender": 1 } }]);
```

3. Count all crimes involving a certain type of weapon;

```
db.weapon.aggregate([
  { $match: { Type: "GIVEN_WEAPON_TYPE" } },
  { $unwind: "$CrimeIds" },
  { $group: { _id: null, crimeCount: { $sum: 1 } } }]);
```

4. Find the crimes and victims of a given district;

```
db.crime.find({ DistrictName: "GIVEN_DISTRICT" }, {
  Id : 1, CrimeType: 1, DateTime: 1, Victim: 1});
```

5. Given an ethnicity find all streets in which that ethnicity has experienced crimes, sorting them by the number of crimes.

```
db.targetvictim.aggregate([
      { $match: { Ethnicity: "GIVEN_ETHNICITY" } },
      { $group: { _id: "$StreetName", crimeCount: { $sum: 1 } } },
      { $sort: { crimeCount: -1 } }]);
```

6. Find the number of cops and all districts where the Crime (Type) "Assault" was done with a weapon of type "FIREARM".

```
db.crime.aggregate([
  {$match: { Type: "Assault", WeaponType: "FIREARM"}},
  {$group: {_id: "$DistrictName", NumCops: { $first: "$NumCops" }}},
  {$project: { DistrictName: "$_id", NumCops: 1,_id: 0}}]);
```

Design in Cassandra

To implement the workload queries in Cassandra, none of the aggregate had to be modified, since any JSON notation can be used.

Crime: Create the type *victims_t* for representing the object with the field of the victim

```
CREATE TYPE victims_t(
    Ethnicity text,
    Age int,
    Gender text
);
```

Since we have disjointed selection attribute over some queries that involves Crime we have 2 different option:

1. create two different table for *Crime* with different primary key, in order to execute directly the query on the partition key

   Table for query Q1:

   ```
   CREATE TABLE CrimeByDistrict (
       Id bigint,
       DateTime timestamp,
       Type text,
       DistrictName text,
       Weapon_type text,
       NumCops int,
       involves list<frozen<victims_t>>
       PRIMARY KEY (DistrictName, DateTime, Id)
   );
   ```

   Table for query Q2 and Q6:

   ```
   CREATE TABLE CrimeByType (
       Id bigint,
       DateTime timestamp,
       Type text,
       DistrictName text,
       Weapon_type text,
       NumCops int,
       involves list<frozen<victims_t>>
       PRIMARY KEY (Type, Weapon_type, Id)
   );
   ```

2. Create 3 index over *Crime* in order to reuse it instead of create 2 tables

   ```
       CREATE TABLE Crime(
           Id bigint,
           DateTime timestamp,
           Type text,
           DistrictName text,
           Weapon_type text,
           NumCops int,
           involves list<frozen<victims_t>>
           PRIMARY KEY (Type,Weapon_type,Id)
   ```

```
);
```

```
CREATE INDEX ON Crime(DistrictName);
CREATE INDEX ON Crime(DateTime);
```

Since create an index over DateTime it's not efficient because the cardinality is very high and, in general, we can achieve better performance having 2 different tables, we select the first option to implement our strategy.

Weapon:

```
CREATE TABLE Weapon(
    Name text,
    Type text,
    CrimeIds set<int>,
    PRIMARY KEY (Type, Name)
);
```

Location: As in the previous case, we have to create 2 types of object for representing the list type of the table Location

```
CREATE TYPE crime_t(
    Id bigint,
    Datetime timestamp,
    Type text,
    SubType text
);

CREATE TYPE targetvictims_t(
    Id bigint,
    Age int,
    Gender text,
    Ethnicity text
);
```

```
CREATE TABLE Location(
    DistrictName text,
    StreetName text,
    occurred_at list<frozen<crime_t>>,
    involves list<frozen<targetvictims_t>>,
    PRIMARY KEY (DistrictName, StreetName)
);
```

TargetVictim:

```
CREATE TABLE TargetVictim(
    Id bigint,
    Ethnicity text,
    CrimeId int,
    StreetName text,
    PRIMARY KEY(Ethnicity, Id)
);
```

Queries:

1. Find all crimes and their type that occurred in a given district and in a given time interval;

```
SELECT Id, Type
FROM CrimeByDistrict
WHERE DistrictName = 'GIVEN_DISTRICT' AND DateTime > START_DATE AND
    DateTime < END_DATE
```

2. Identify all victims given a crime type;

```
SELECT Age, Gender, Ethnicity
FROM CrimeByType
WHERE Type = 'GIVEN_TYPE'
```

3. Count all crimes involving a certain type of weapon;

   We can't count the number of crimes because inside the table Weapon the crime are represented as a set of int. Cassandra doesn't allow to count over a set.

   For solving this problem we can query over the table CrimeByType and filter over the weapon type inside it Since the Weapon_type is not the partition key, we have to create an index for execute this query.

```
CREATE INDEX ON CrimeByType(Weapon_type)
```

```
SELECT COUNT(*)
FROM CrimeByType
WHERE Weapon_type = 'GIVEN_WEAPON_TYPE'
```

4. Find the crimes and victims of a given district;

```
SELECT occurred_at, involves
FROM Location
WHERE DistrictName = 'GIVEN_DISTRICT'
```

5. Given an ethnicity find all streets in which that ethnicity has experienced crimes, sorting them by the number of crimes.

   In this query we have a problem that is similar to the query 3, although COUNT is available in Cassandra, it does not support direct aggregation as a grouped count by StreetName. To do so we can return a partial result that can be grouped and sorted at Application level

```
SELECT StreetName, Id
FROM TargetVictim
WHERE Ethnicity = 'GIVEN_ETHNICITY'
```

6. Find the number of cops and all districts where the Crime (Type) "Assault" was done with a weapon of type "FIREARM".

```
SELECT DistrictName, NumCops
FROM CrimeByType
WHERE Type = 'ASSAULT' AND Weapon_type = "FIREARM"
```

Design in Neo4J

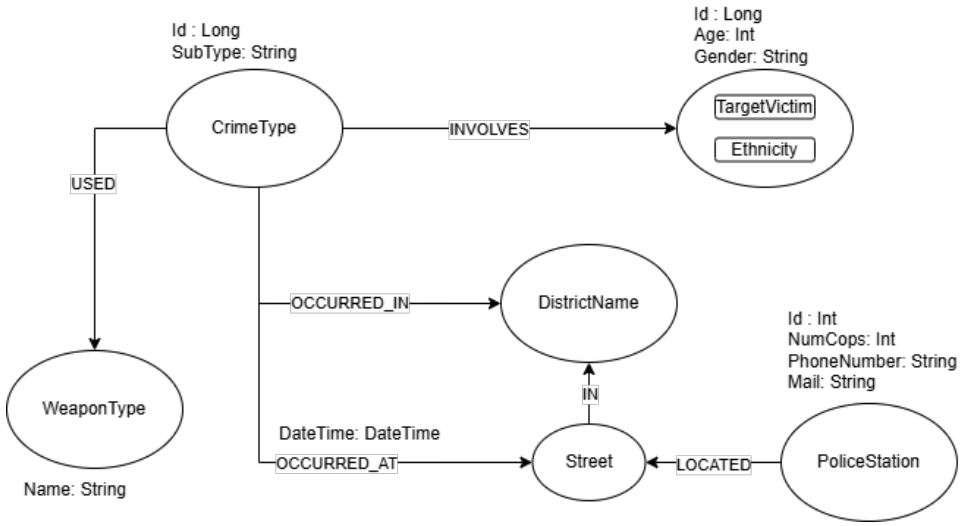

Figure 3: Graph Structure

Nodes and Relationships in the Graph Model

The graph-based representation comprises the following nodes and relationships, each defined to facilitate efficient querying and analysis:

Nodes

- **CrimeType:** Each node represents a specific type of crime, with the label being a string containing the crime type's name. This label was chosen to enhance queries related to crime types. These nodes also include details such as *Crime ID* and *Sub Type*. Multiple nodes with the same label may exist, corresponding to the number of crimes in the dataset.
- **WeaponType:** Each node represents a type of weapon, with the label containing the weapon type's name. This label facilitates queries related to weapon types. These nodes also store information such as *Weapon Name*. While the addition of new nodes is possible over time, modifications to existing nodes are rare.
- **Target Victim and Ethnicity:** *TargetVictim* nodes represent individual victims, while the *Ethnicity* label specifies the victim's ethnicity (e.g., Asian, Hispanic). These nodes include attributes such as *Victim ID*, *Age*, and *Gender*.
- **DistrictName:** Each node represents one of the 21 districts in Los Angeles, labeled with the district name. This design choice supports improved querying of district-related information.
- **Street:** These nodes represent streets within Los Angeles, labeled with their names. Like the district nodes, this choice enhances queries related to street-level details.
- **PoliceStation:** These nodes encapsulate information about police stations within Los Angeles, including attributes such as *Station ID*, *Number of Officers*, *Phone Number*, and *Email Address*.

Relationships

- **USED:** Indicates the use of a weapon in a crime.
- **INVOLVES:** Specifies the involvement of a victim in a crime. Only persons, not objects, can be victims.
- **OCCURRED_IN:** Represents a less specific location of the crime scene.
- **OCCURRED_AT:** Specifies a more precise location of the crime scene, including the time of occurrence.
- **IN:** Connects a street to its corresponding district.
- **LOCATED:** Indicates that a police station is situated on a specific street.

Queries

1. Find all crimes and their type that occurred in a given district and in a given time interval;

```
MATCH (c)-[oa:OCCURRED_AT]->(s), (s)-[i:IN]->(d)
WHERE d:District:Devonshire AND c:CrimeType
AND oa.DateTime >= datetime("2020-11-01T00:00:00Z")
AND oa.DateTime <= datetime("2020-11-30T23:59:59Z")
RETURN
  c.Type AS CrimeType, c.CrimeId AS CrimeId,
  oa.DateTime AS DateTime, d.Name AS District
```

2. Identify all victims given a crime type;

```
MATCH (c)-[:INVOLVES]->(v:TargetVictim)
WHERE c:CrimeType:ASSAULT
RETURN v.VictimId AS VictimId, v.Age AS VictimAge, v.Gender AS
VictimGender, v.Ethnicity AS VictimEthnicity
```

3. Count all crimes involving a certain type of weapon;

```
MATCH (c)-[:USED]->(w)
WHERE w:FIREARM
RETURN COUNT(c) AS CrimeCount
```

4. Find the crimes and victims of a given district;

```
MATCH (c)-[:OCCURRED_IN]->(d:District:Central),
(c)-[:INVOLVES]->(v:TargetVictim)
RETURN c.Type AS CrimeType, c.CrimeId AS CrimeId, v.VictimId AS
   VictimId,
v.Age AS VictimAge, v.Gender AS VictimGender, v.Ethnicity AS Ethnicity
```

5. Given an ethnicity find all streets in which that ethnicity has experienced crimes, sorting them by the number of crimes.

```
MATCH (c)-[:INVOLVES]->(v:TargetVictim:Hispanic),
(c)-[:OCCURRED_AT]->(s)
RETURN s.Name AS StreetName, COUNT(c) AS CrimeCount
ORDER BY CrimeCount DESC
```

6. Find the number of cops and all districts where the Crime (Type) "Assault" was done with a weapon of type "FIREARM".

```
MATCH
(c:CrimeType:ASSAULT)-[:OCCURRED_IN]->(d:District),
(c:CrimeType:ASSAULT)-[:USED]->(w:WeaponType:FIREARM),
(ps:PoliceStation)-[:LOCATED]->(s:Street),
(s:Street)-[:IN]->(d:District)
RETURN DISTINCT d.Name AS DistrictName, ps.NumCops AS NumberOfCops
```

Most Suitable System among the designed ones

Given the characteristics of our application:

1. Predominantly read-intensive workloads.
2. High availability as a critical requirement.
3. Some consistency constraints, particularly for certain operations.
4. No need for batch processing.
5. Highly relational queries to address the interconnected nature of our data

We have selected **Neo4j** as the most suitable backend system for our application. This decision is driven by the following considerations:

1. **Graph-Centric Model**: The core of our application involves relationships and traversals, such as linking crimes to districts, victims, and weapons. These requirements align perfectly with the advantages of graph databases, where Neo4j excels in relationship-centric queries and analysis.
2. **Efficient Query Language**: Neo4j's Cypher query language is well-suited to our workload, providing a more intuitive and efficient way to handle relational queries compared to systems like MongoDB.
3. **Directly Reflecting the Conceptual Schema**: Neo4j's graph model allows us to closely mirror the conceptual schema of our application. This is in contrast to systems like Cassandra, which require schema designs that are tailored to specific queries, leading to potential rigidity.
4. **Optimized for Read-Intensive Workloads**: Neo4j's architecture supports high availability and read performance, making it ideal for our application's predominantly read-heavy operations.
5. **Low Write-Workloads**: The write operations of the system are very infrequent, on average less than 700 write per day. With this number of writes Neo4j it's more than enough to handle it.
6. **Consistency Guarantees**: Neo4j offers sufficient consistency for our needs, providing strong consistency for critical operations while maintaining availability.
7. **CA-Oriented Design**: Neo4j's CA (Consistency and Availability) guarantees fit our requirements, as network partitions are less of a concern within our controlled deployment.

In addition to that, Neo4J became a good choice also for the fact that if we want to represent our dataset as a map (so the graph is composed by connection among the street with weights on the relations) we can easily achieve in Neo4j simply adding all the needed relation maintaining as is the structure and the relations of all the other nodes.

We decided to discard the other systems for this reasons:

1. **MongoDB**:
   (a) **Unintuitive Relationship Handling**: MongoDB struggles with efficiently managing highly relational queries. If the future workloads will include topological info MongoDB is not suitable for this kind of queries.
2. **Cassandra**:

(a) **Rigid Schema Requirements**: Cassandra requires denormalized tables tailored to specific queries, which conflicts with the dynamic and diverse query scenarios of our application.

(b) **Lack of Query Flexibility**: Cassandra's design is ill-suited for future workload changes, which are likely in the dynamic context of crime analysis.

System Configuration for Neo4j

To ensure the efficient storage and processing of data for our application, we propose the following system configuration for deploying Neo4j as the backend. The configuration is tailored to the application's requirements, including high availability, read-intensive workloads, and relational data queries.

Neo4j will be deployed in a **Causal Cluster** configuration to meet the high availability and consistency requirements:

1. **One Leader Node**: Handles all write operations to ensure strong consistency.
2. **Two-Four Follower Nodes**: Replicate data asynchronously for read scalability and redundancy.
3. **One Read Replica (optional)**: To offload intensive read operations and ensure scalability for future demands.

This setup ensures high availability, as followers can elect a new leader in the event of node failure.

In addition the following parameter can be choosen to meet the requirements:

1. Enable cluster mode with causal consistency using

   ```
   dbms.mode=CORE.
   ```

2. Configure fault tolerance with this parameter to ensure quorum.

   ```
   causal_clustering.minimum_core_cluster_size_at_runtime=3
   ```

3. Use query logging to monitor and optimize frequently run queries.

   ```
   dbms.logs.query.enabled=true
   ```

4. Use constraints:

   (a) Enforce unique constraints where applicable (e.g., Crime ID, District Name).

Since the application requires the regular addition of small datasets (e.g., daily crime records), we can use batch import tools (e.g., *neo4j-admin import*) for initial data load and, for incremental updates, implement ETL pipelines using APOC (Awesome Procedures on Cypher) to load and transform new records.

For maintenance and disaster recovery, we can also implement periodic backups using *neo4j-admin backup*

Since Neo4j is schema-less we don't have to provide any schema details.

Selected system logical schema

Neo4J's lack of a predefined schema eliminates the need to specify structural details, such as table definitions required in systems like Cassandra. Instead, we can directly and dynamically create nodes and establish relationships, as we will illustrate in the next step. Here is a simple example demonstrating how the nodes and their relationships should look during the database insertion process:

```
CREATE (:District:Downtown {name: 'Downtown'});
CREATE (:Street:VINELAND {Name: 'VINELAND'});
CREATE (:CrimeType:Burglary {type: 'THEFT', crimeId: 554196, subType:'THEFT
    PLAIN - PETTY'});
CREATE (:TargetVictim:Hispanic {VictimId: 604346, Ethnicity: 'Hispanic', age:
    34, gender: 'M'});
CREATE (:WeaponType:FIREARM {type: 'FIREARM', Name: 'HAND GUN'});
CREATE (:PoliceStation {PoliceStationId: 2, Mail: 'susanmoore@example.com',
    NumCops: 264, PhoneNumber: '(323) 543-8286'});

MATCH (d:District:Downtown),
      (s:Street:VINELAND),
      (c:CrimeType {crimeId: 554196}),
      (v:TargetVictim {VictimId: 604346}),
      (w:WeaponType:FIREARM {Name: 'HAND GUN'}),
      (p:PoliceStation {PoliceStationId: 2})
CREATE (c)-[:OCCURRED_IN]->(d);
CREATE (c)-[:OCCURRED_AT {DateTime : '2020-11-09T23:00:00Z'}]->(s);
CREATE (s)-[:IN]->(d);
CREATE (c)-[:INVOLVES]->(v);
CREATE (c)-[:USED]->(w);
CREATE (p)-[:LOCATED]->(s);
```

Selected system instance creation

For our system, we utilized two primary datasets: one detailing crime incidents and another containing information about LAPD police stations. These datasets were preprocessed to ensure data quality, and additional synthetic data was generated using a Python script to enrich the information. Subsequently, the datasets were divided into multiple sub-datasets optimized for import into the Neo4J Aura platform.

The original datasets can be accessed at the following links:

1. Crime Data from 2020 to Present
2. LAPD Police Stations Dataset

However, due to Neo4j Aura free tier limit of nodes and edges count, we only be able to insert a very small fraction of our datasets, around 95K nodes and 141K relationships. Even under these constraints we inserted around 8MB.

Below, we present the Cypher script utilized to import the data into the Neo4J database:

```
:param {
  file_path_root: 'https://raw.githubusercontent.com/Emulars/
  adm-project/refs/heads/main/Neo4J/to_import/',
  file_0: 'DistrictName.csv',
  file_1: 'TargetVictim.csv',
  file_2: 'CrimeType.csv',
  file_3: 'WeaponType.csv',
  file_4: 'Street.csv',
  file_5: 'PoliceStation.csv',
  file_6: 'CrimeType_OccurredIn_DistrictName.csv',
  file_7: 'CrimeType_Involves_TargetVictim.csv',
  file_8: 'WeaponType_Used_CrimeType.csv',
  file_9: 'CrimeType_OccurredAt_Street.csv',
  file_10: 'Street_In_District.csv',
  file_11: 'Street_Located_PoliceStation.csv'
};

// CONSTRAINT creation
// -------------------
CREATE CONSTRAINT `CrimeId_CrimeType_uniq` IF NOT EXISTS
    FOR (n: `CrimeType`)
    REQUIRE (n.`CrimeId`) IS UNIQUE;
CREATE CONSTRAINT `Name_WeaponType_uniq` IF NOT EXISTS
    FOR (n: `WeaponType`)
    REQUIRE (n.`Name`) IS UNIQUE;
CREATE CONSTRAINT `VictimId_TargetVictim_uniq` IF NOT EXISTS
    FOR (n: `TargetVictim`)
    REQUIRE (n.`VictimId`) IS UNIQUE;
CREATE CONSTRAINT `Name_District_uniq` IF NOT EXISTS
    FOR (n: `District`)
    REQUIRE (n.`Name`) IS UNIQUE;
CREATE CONSTRAINT `Name_Street_uniq` IF NOT EXISTS
    FOR (n: `Street`)
    REQUIRE (n.`Name`) IS UNIQUE;
CREATE CONSTRAINT `PoliceStationId_PoliceStation_uniq` IF NOT EXISTS
    FOR (n: `PoliceStation`)
    REQUIRE (n.`PoliceStationId`) IS UNIQUE;
```

```
:param {idsToSkip: []};

// NODE load
// ---------
LOAD CSV WITH HEADERS FROM ($file_path_root + $file_2) AS row
WITH row
WHERE NOT row.`CrimeId` IN $idsToSkip AND NOT toInteger(trim(row.`CrimeId`))
    IS NULL
CALL {
  WITH row
  MERGE (n: `CrimeType` { `CrimeId`: toInteger(trim(row.`CrimeId`)) })
  SET n.`CrimeId` = toInteger(trim(row.`CrimeId`))
  SET n.`Type` = row.`Type`
  SET n.`Subtype` = row.`Subtype`
} IN TRANSACTIONS OF 10000 ROWS;

LOAD CSV WITH HEADERS FROM ($file_path_root + $file_3) AS row
WITH row
WHERE NOT row.`Name` IN $idsToSkip AND NOT row.`Name` IS NULL
CALL {
  WITH row
  MERGE (n: `WeaponType` { `Name`: row.`Name` })
  SET n.`Name` = row.`Name`
  SET n.`Type` = row.`Type`
} IN TRANSACTIONS OF 10000 ROWS;

LOAD CSV WITH HEADERS FROM ($file_path_root + $file_1) AS row
WITH row
WHERE NOT row.`VictimId` IN $idsToSkip AND NOT toInteger(trim(row.`VictimId`))
    IS NULL
CALL {
  WITH row
  MERGE (n: `TargetVictim` { `VictimId`: toInteger(trim(row.`VictimId`)) })
  SET n.`VictimId` = toInteger(trim(row.`VictimId`))
  SET n.`Age` = toInteger(trim(row.`Age`))
  SET n.`Gender` = row.`Gender`
  SET n.`Ethnicity` = row.`Ethnicity`
} IN TRANSACTIONS OF 10000 ROWS;

LOAD CSV WITH HEADERS FROM ($file_path_root + $file_0) AS row
WITH row
WHERE NOT row.`Name` IN $idsToSkip AND NOT row.`Name` IS NULL
CALL {
  WITH row
  MERGE (n: `District` { `Name`: row.`Name` })
  SET n.`Name` = row.`Name`
} IN TRANSACTIONS OF 10000 ROWS;

LOAD CSV WITH HEADERS FROM ($file_path_root + $file_4) AS row
WITH row
WHERE NOT row.`Name` IN $idsToSkip AND NOT row.`Name` IS NULL
CALL {
  WITH row
  MERGE (n: `Street` { `Name`: row.`Name` })
```

```
  SET n.`Name` = row.`Name`
} IN TRANSACTIONS OF 10000 ROWS;

LOAD CSV WITH HEADERS FROM ($file_path_root + $file_5) AS row
WITH row
WHERE NOT row.`PoliceStationId` IN $idsToSkip AND NOT
    toInteger(trim(row.`PoliceStationId`)) IS NULL
CALL {
  WITH row
  MERGE (n: `PoliceStation` { `PoliceStationId`:
      toInteger(trim(row.`PoliceStationId`)) })
  SET n.`PoliceStationId` = toInteger(trim(row.`PoliceStationId`))
  SET n.`PhoneNumber` = row.`PhoneNumber`
  SET n.`Mail` = row.`Mail`
  SET n.`NumCops` = toInteger(trim(row.`NumCops`))
} IN TRANSACTIONS OF 10000 ROWS;


// RELATIONSHIP load
// -----------------

LOAD CSV WITH HEADERS FROM ($file_path_root + $file_8) AS row
WITH row
CALL {
  WITH row
  MATCH (source: `CrimeType` { `CrimeId`: toInteger(trim(row.`CrimeId`)) })
  MATCH (target: `WeaponType` { `Name`: row.`Weapon` })
  MERGE (source)-[r: `USED`]->(target)
} IN TRANSACTIONS OF 10000 ROWS;

LOAD CSV WITH HEADERS FROM ($file_path_root + $file_7) AS row
WITH row
CALL {
  WITH row
  MATCH (source: `CrimeType` { `CrimeId`: toInteger(trim(row.`CrimeId`)) })
  MATCH (target: `TargetVictim` { `VictimId`: toInteger(trim(row.`VictimId`))
      })
  MERGE (source)-[r: `INVOLVES`]->(target)
} IN TRANSACTIONS OF 10000 ROWS;

LOAD CSV WITH HEADERS FROM ($file_path_root + $file_6) AS row
WITH row
CALL {
  WITH row
  MATCH (source: `CrimeType` { `CrimeId`: toInteger(trim(row.`CrimeId`)) })
  MATCH (target: `District` { `Name`: row.`Name` })
  MERGE (source)-[r: `OCCURRED_IN`]->(target)
} IN TRANSACTIONS OF 10000 ROWS;

LOAD CSV WITH HEADERS FROM ($file_path_root + $file_9) AS row
WITH row
CALL {
  WITH row
  MATCH (source: `CrimeType` { `CrimeId`: toInteger(trim(row.`CrimeId`)) })
  MATCH (target: `Street` { `Name`: row.`StreetId` })
```

```
  MERGE (source)-[r: `OCCURRED_AT`]->(target)
} IN TRANSACTIONS OF 10000 ROWS;

LOAD CSV WITH HEADERS FROM ($file_path_root + $file_10) AS row
WITH row
CALL {
  WITH row
  MATCH (source: `Street` { `Name`: row.`Street` })
  MATCH (target: `District` { `Name`: row.`District` })
  MERGE (source)-[r: `IN`]->(target)
} IN TRANSACTIONS OF 10000 ROWS;

LOAD CSV WITH HEADERS FROM ($file_path_root + $file_11) AS row
WITH row
CALL {
  WITH row
  MATCH (source: `PoliceStation` { `PoliceStationId`:
      toInteger(trim(row.`PoliceStationId`)) })
  MATCH (target: `Street` { `Name`: row.`Street` })
  MERGE (source)-[r: `LOCATED`]->(target)
} IN TRANSACTIONS OF 10000 ROWS;

// IMPROVED LABELS
// ----------------

// Add labels to CrimeType nodes
MATCH (c:CrimeType)
CALL apoc.create.addLabels(c, [c.Type]) YIELD node
RETURN node

// Add labels to WeaponType nodes
MATCH (w:WeaponType)
CALL apoc.create.addLabels(w, [w.Type]) YIELD node
RETURN node

// Add labels to District nodes
MATCH (d:District)
CALL apoc.create.addLabels(d, [d.Name]) YIELD node
RETURN node

// Add labels to Street nodes
MATCH (s:Street)
CALL apoc.create.addLabels(s, [s.Name]) YIELD node
RETURN node

// Add labels to TargetVictim nodes
MATCH (t:TargetVictim)
CALL apoc.create.addLabels(t, [t.Ethnicity]) YIELD node
RETURN node
```

NOTICE:

This type of import does not support assigning the DateTime property directly to the OCCURRED_AT relationship.

Selected System implemented workload

Find all crimes and their type that occurred in a given district and in a given time interval;

```
MATCH (c)-[oa:OCCURRED_AT]->(s), (s)-[i:IN]->(d)
WHERE d:Central AND c:CrimeType
AND oa.DateTime >= datetime("2020-11-01T00:00:00Z")
AND oa.DateTime <= datetime("2020-11-30T23:59:59Z")
RETURN
c.Type AS CrimeType, c.CrimeId AS CrimeId
// Returned 45 records after 98 ms and completed after 127 ms
```

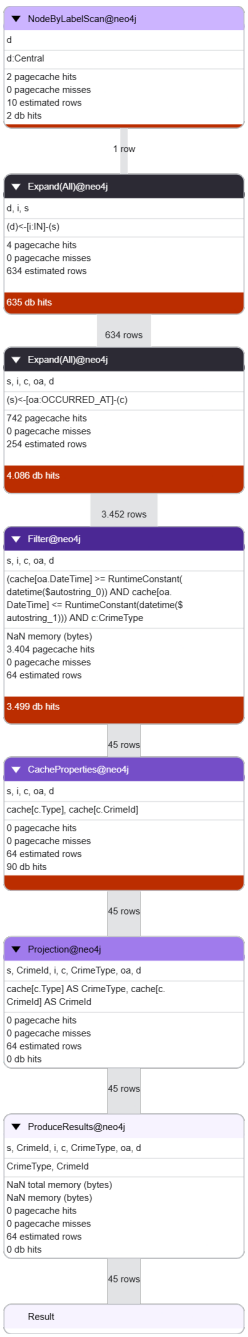

Figure 4: ProfileResults

Identify all victims given a crime type;

```
MATCH (c)-[:INVOLVES]->(v:TargetVictim)
WHERE c:ASSAULT
RETURN v.VictimId AS VictimId, v.Age AS VictimAge, v.Gender AS
VictimGender, v.Ethnicity AS VictimEthnicity
// Returned 5000 records after 23 ms and completed after 62 ms.
```

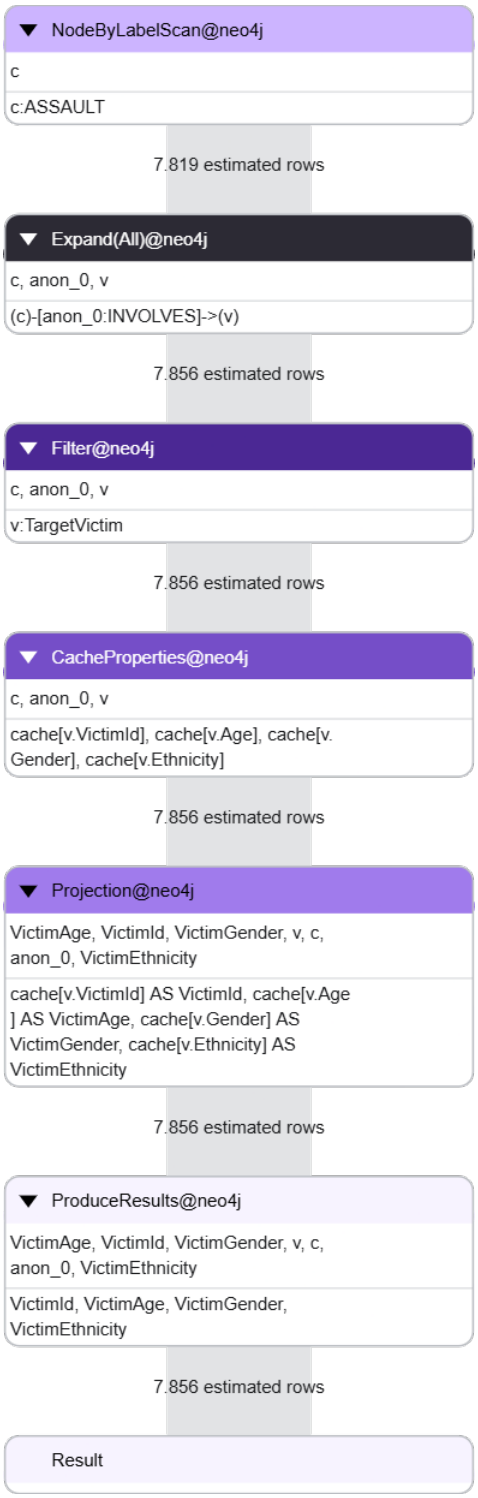

Figure 5: Explain Results

Count all crimes involving a certain type of weapon;

```
MATCH (c)-[:USED]->(w)
WHERE w:FIREARM
RETURN COUNT(c) AS CrimeCount
// Returned 1 record after 38 ms and completed after 38 ms
```

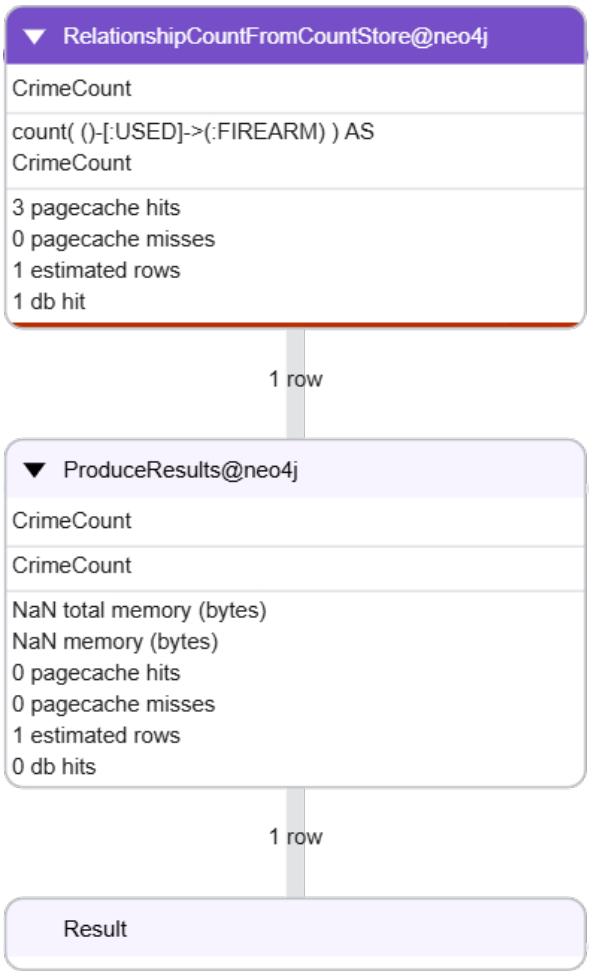

Figure 6: Profile Results

Find the crimes and victims of a given district;

```
MATCH (c)-[:OCCURRED_IN]->(d:Central),
(c)-[:INVOLVES]->(v:TargetVictim)
RETURN c.Type AS CrimeType, c.CrimeId AS CrimeId, v.VictimId AS VictimId,
v.Age AS VictimAge, v.Gender AS VictimGender, v.Ethnicity AS Ethnicity
// Returned 2662 records after 69 ms and completed after 82 ms
```

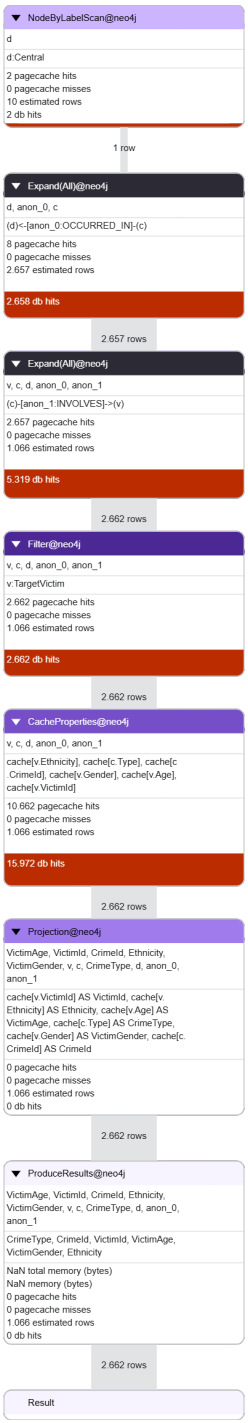

Figure 7: Profile Results

Given an ethnicity find all streets in which that ethnicity has experienced crimes, sorting them by the number of crimes.

```
MATCH (c)-[:INVOLVES]->(v:Hispanic), (c)-[:OCCURRED_AT]->(s)
RETURN s.Name AS StreetName, COUNT(c) AS CrimeCount
ORDER BY CrimeCount DESC
// Returned 5000 records after 72 ms and completed after 186 ms
```

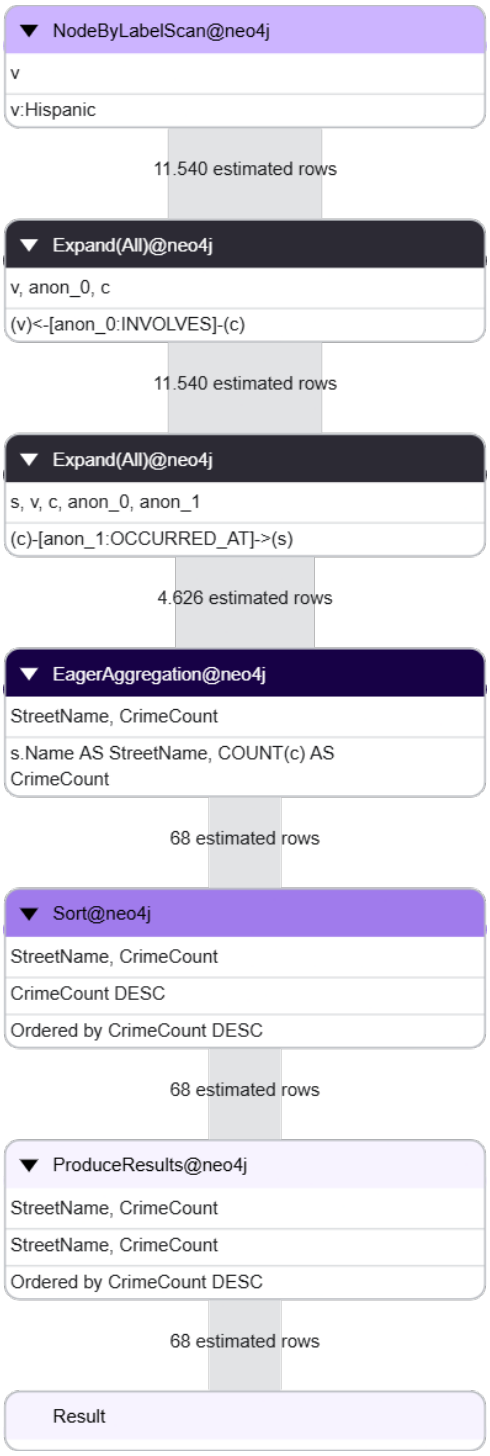

Figure 8: Explain Results

Find the number of cops and all districts where the Crime (Type) "Assault" was done with a weapon of type "FIREARM".

```
MATCH
(c:ASSAULT)-[:OCCURRED_IN]->(d:District),
(c:ASSAULT)-[:USED]->(w:FIREARM),
(ps:PoliceStation)-[:LOCATED]->(s:Street),
(s:Street)-[:IN]->(d:District)
RETURN DISTINCT d.Name AS DistrictName, ps.NumCops AS NumberOfCops
// Returned 21 records after 236 ms and completed after 2 ms.
```

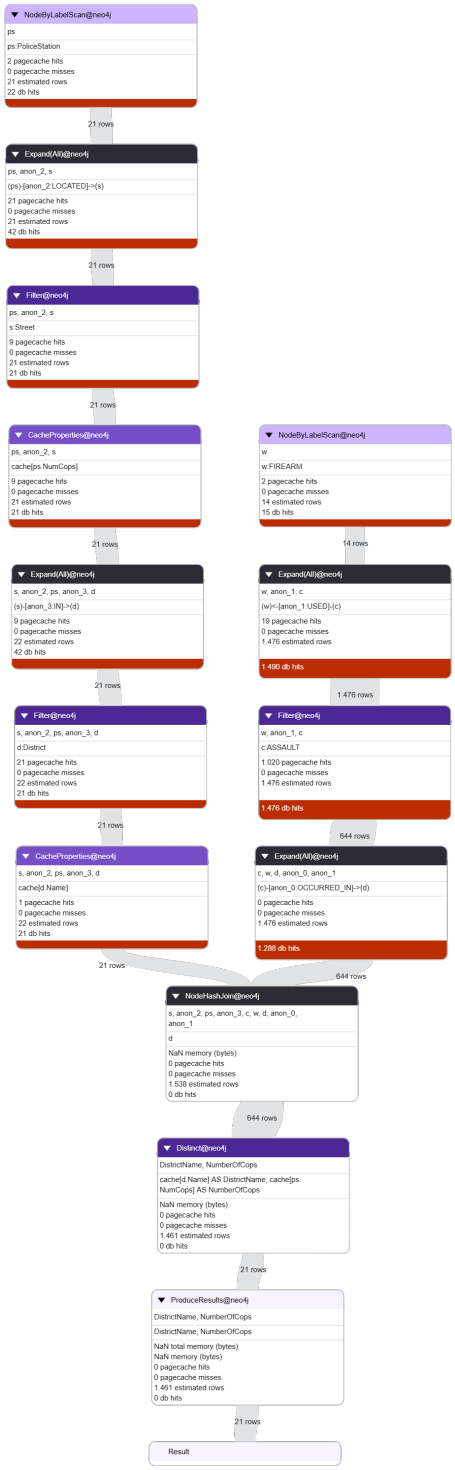

Figure 9: Profile Results

NOTICE:

Since our workload queries work directly with label and not with properties we decided to not use indexes because Neo4J is already optimized for queries over labels. The only index that we tried to implement is the one over DateTime but, executing the query with PROFILE we noticed that the index was not used.

RDFS/OWL and RDF

The following model represents the main entities, relationships, and constraints of our schema in RDFS/OWL, with attention to domains, ranges, inverses, equivalences, disjoint classes, and functional properties.

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix ex: <http://example.org/ontology#> .

### Classes
ex:Crime a owl:Class .
ex:Victim a owl:Class .
ex:Weapon a owl:Class .
ex:District a owl:Class .
ex:Street a owl:Class .
ex:PoliceStation a owl:Class .
ex:StreetName a owl:Class .

### Object Properties
ex:hasVictim a owl:ObjectProperty ;
    rdfs:domain ex:Crime ;
    rdfs:range ex:Victim ;
    owl:inverseOf ex:isVictimOf .

ex:usesWeapon a owl:ObjectProperty ;
    rdfs:domain ex:Crime ;
    rdfs:range ex:Weapon ;
    owl:inverseOf ex:isWeaponOf .

ex:occurredIn a owl:ObjectProperty ;
    rdfs:domain ex:Crime ;
    rdfs:range ex:District .

ex:occurredAt a owl:ObjectProperty ;
    rdfs:domain ex:Crime ;
    rdfs:range ex:Street .

ex:isInDistrict a owl:ObjectProperty ;
    rdfs:domain ex:Street ;
    rdfs:range ex:District ;
    owl:inverseOf ex:hasStreet .

ex:HasStreetName a owl:ObjectProperty ;
    rdfs:domain ex:Street ;
    rdfs:range ex:StreetName .

### Datatype Properties
ex:hasNumCops a owl:DatatypeProperty ;
    rdfs:domain ex:PoliceStation ;
    rdfs:range xsd:integer .
```

```
### Disjoint Classes
[] a owl:AllDisjointClasses ; owl:members (ex:Victim ex:Weapon) .
[] a owl:AllDisjointClasses ; owl:members (ex:Crime ex:PoliceStation) .
### Functional and Inverse Functional Properties
ex:hasNumCops a owl:FunctionalProperty .
ex:occurredAt a owl:FunctionalProperty .
ex:HasStreetName a owl:InverseFunctionalProperty
```

Now we model in RDF some instances relating to the corresponding class/property, clarifying which individuals are identical or different

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix ex: <http://example.org/ontology#> .

### Instances of Classes
ex:Crime1 a ex:Crime ;
    ex:occurredIn ex:District1 ;
    ex:hasType "Burglary"^^xsd:string ;
    ex:hasDateTime "2024-12-31T23:59:59"^^xsd:dateTime .

ex:Crime2 a ex:Crime ;
    ex:occurredIn ex:District1 ;
    ex:hasType "Robbery"^^xsd:string ;
    ex:hasDateTime "2025-01-15T12:30:00"^^xsd:dateTime .

ex:Crime3 a ex:Crime ;
    ex:occurredIn ex:District1 ;
    ex:hasType "Assault"^^xsd:string ;
    ex:hasDateTime "2025-01-25T18:00:00"^^xsd:dateTime .

ex:Victim1 a ex:Victim ;
    ex:hasEthnicity "Hispanic"^^xsd:string ;
    ex:hasAge "30"^^xsd:integer ;
    ex:hasGender "Female"^^xsd:string ;
    ex:isVictimOf ex:Crime2 .

ex:Weapon1 a ex:Weapon ;
    ex:hasName "Pistol"^^xsd:string ;
    ex:isWeaponOf ex:Crime2 .

ex:District1 a ex:District ;
    ex:hasName "Downtown"^^xsd:string ;
    ex:hasStreet ex:Street1 .

ex:Street1 a ex:Street ;
    ex:hasName "Main Street"^^xsd:string ;
    ex:isInDistrict ex:District1 .

ex:PoliceStation1 a ex:PoliceStation ;
    ex:hasNumCops "50"^^xsd:integer ;
    ex:locatedAt ex:Street1 .
```

```
### Individuals Declared as Identical
ex:AreaDowntown
    owl:sameAs ex:District1 .

### Individuals Declared as Different
[] a owl:AllDifferent ;
    owl:members (ex:Victim1 ex:Weapon1 ex:PoliceStation1) .


### Adding crimes with Hispanic victims and occurrence at specific streets
ex:Crime5 a ex:Crime ;
    ex:occurredAt ex:Street1 ;
    ex:hasVictim ex:Victim2 ;
    ex:hasType "Mugging"^^xsd:string ;
    ex:hasDateTime "2025-01-18T09:30:00"^^xsd:dateTime .

ex:Crime6 a ex:Crime ;
    ex:occurredAt ex:Street1 ;
    ex:hasVictim ex:Victim3 ;
    ex:hasType "Robbery"^^xsd:string ;
    ex:hasDateTime "2025-01-22T14:45:00"^^xsd:dateTime .

ex:Crime7 a ex:Crime ;
    ex:occurredAt ex:Street2 ;
    ex:hasVictim ex:Victim4 ;
    ex:hasType "Theft"^^xsd:string ;
    ex:hasDateTime "2025-01-23T11:15:00"^^xsd:dateTime .

ex:Crime8 a ex:Crime ;
    ex:occurredAt ex:Street2 ;
    ex:hasVictim ex:Victim5 ;
    ex:hasType "Assault"^^xsd:string ;
    ex:hasDateTime "2025-01-24T19:00:00"^^xsd:dateTime .

# Adding new streets
ex:Street2 a ex:Street ;
    ex:hasName "Broadway"^^xsd:string ;
    ex:isInDistrict ex:District1 .
```

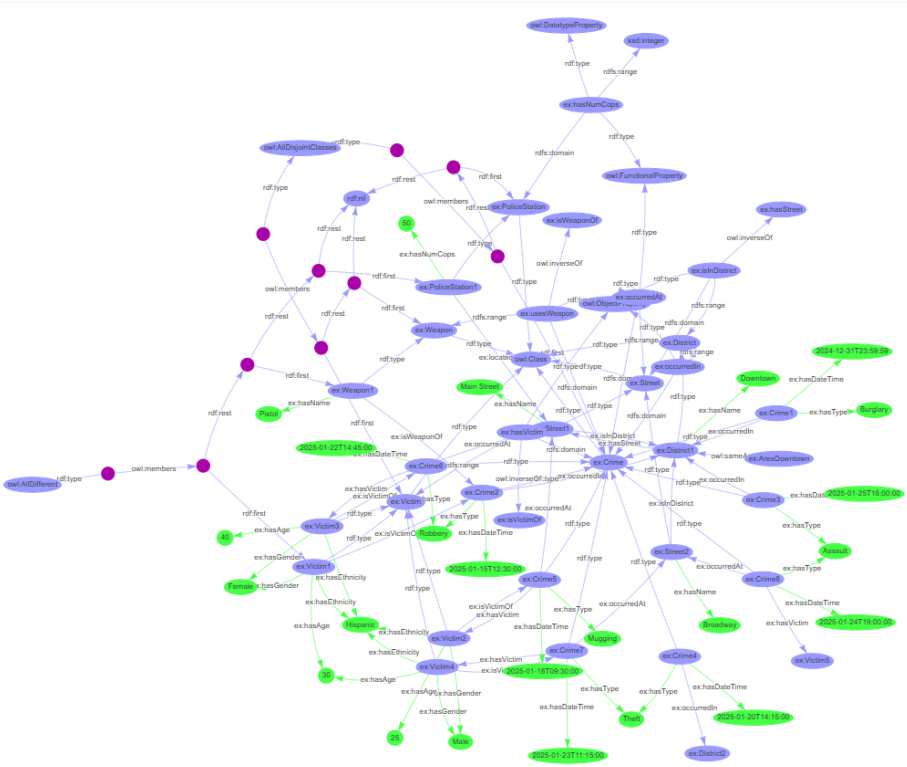

Figure 10: Resulting Graph

SPARQL queries

Now we specify some queries over the RDF defined dataset, strating from our workload defined before:

1. Find all crimes and their type that occurred in a given district and in a given time interval

```
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX ex: <http://example.org/ontology#>

SELECT
    (STRAFTER(STR(?crime), "#") AS ?crimeValue)
    ?type
    (STR(?datetime) AS ?datetimeValue)
WHERE {
    ?crime a ex:Crime ;
        ex:occurredIn ?district ;
        ex:hasType ?type ;
        ex:hasDateTime ?datetime .
    ?district ex:hasName ?distrName.
    FILTER (?datetime >= "2025-01-01T00:00:00"^^xsd:dateTime &&
    ?datetime <= "2025-01-31T23:59:59"^^xsd:dateTime &&
    ?distrName= "Downtown")
}
```

crimeValue	type	datetimeValue
"Crime3"	"Assault"	"2025-01-25T18:00:00"
"Crime2"	"Robbery"	"2025-01-15T12:30:00"

2. Identify all victims of a specific crime type

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ex: <http://example.org/ontology#>

CONSTRUCT {
  ?victim a ex:Victim ;
          ex:isVictimOf ?crime ;
          ex:hasEthnicity ?ethnicity ;
          ex:hasAge ?age ;
          ex:hasGender ?gender .
}
WHERE {
  ?crime a ex:Crime ;
         ex:hasType "Robbery" .
  ?victim a ex:Victim ;
          ex:isVictimOf ?crime ;
          ex:hasEthnicity ?ethnicity ;
          ex:hasAge ?age ;
          ex:hasGender ?gender .
}
```

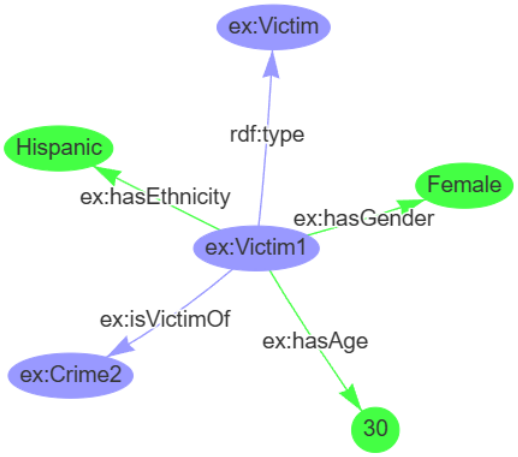

Figure 11: Query 2 Result

3. Given an ethnicity, find all streets where that ethnicity has experienced crimes, sorted by the number of crimes

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ex: <http://example.org/ontology#>

SELECT (STRAFTER(STR(?street ), "#") AS ?Street) (COUNT(?crime) AS
    ?crimeCount)
WHERE {
  ?crime a ex:Crime ;
         ex:occurredAt ?street ;
         ex:hasVictim ?victim .
  ?victim ex:hasEthnicity "Hispanic" .
}
GROUP BY ?street
ORDER BY DESC(?crimeCount)
```

Street	crimeCount
"Street1"	2
"Street2"	1