Universidad Diego Portales

Facultad de Ingeniería
Escuela de Informática y Telecomunicaciones

Tarea 2 Sistemas Distribuidos

Alumno:
Kevin Cabrera

junio 2025

Contents

1 Introducción

Este informe detalla el desarrollo de un sistema distribuido que simula la recolección, almacenamiento y consulta eficiente de eventos de tráfico en una ciudad, utilizando herramientas modernas como Python, MongoDB, Apache Pig y Docker. Inspirado en el sistema de Waze, este proyecto busca implementar un sistema modular, automatizado y escalable que permita capturar al menos 10.000 eventos de tráfico, almacenarlos en una base de datos y realizar consultas optimizadas mediante políticas de caché.

En esta segunda fase del proyecto, se incorpora un módulo de procesamiento distribuido utilizando Apache Pig, el cual permite limpiar, transformar, agrupar y analizar los eventos de tráfico simulados, proporcionando una capa de análisis exploratorio clave para la toma de decisiones viales. Esto representa un avance significativo respecto a la primera entrega, al pasar desde la simulación y evaluación de rendimiento hacia la estructuración y análisis de los datos recolectados.

2 Desarrollo del Sistema

2.1 Componentes Principales

El sistema fue construido de forma modular y cuenta con los siguientes componentes:

- **Generador de eventos**: Simula 10.000 eventos de tráfico con distintos tipos (accidentes, congestión, policía, peligro) y ubicaciones aleatorias.
- **Generador de tráfico**: Crea consultas hacia los eventos simulados, siguiendo distribuciones estadísticas Poisson y Binomial, totalmente ajustables desde el módulo `config.py`.
- **Evaluador**: Mide el rendimiento del sistema bajo diferentes políticas de caché (LRU y LFU), cantidades de consultas y tamaños de caché, generando gráficos comparativos automáticamente.
- **Caché**: Se implementaron las políticas de reemplazo LRU y LFU utilizando la librería `cachetools`.
- **Base de datos**: MongoDB fue utilizado como sistema de almacenamiento no relacional para los eventos.

2.2 Contenerización y ejecución

El sistema completo está contenerizado mediante `Docker` y se levanta con el comando:

```
docker-compose up --build
```

Este comando crea una red interna entre dos contenedores: uno para MongoDB y otro para el sistema en Python. La comunicación se realiza mediante una variable de entorno (`MONGO_URL`).

2.3 Procesamiento distribuido con Apache Pig

Para esta segunda fase, se añadió un módulo de procesamiento distribuido utilizando Apache Pig, encargado de transformar los datos simulados en estructuras analíticas útiles. Este módulo fue diseñado como un pipeline que opera sobre el archivo CSV generado por el sistema en Python y realiza operaciones tipo MapReduce.

Las tareas implementadas fueron:

- **Filtrado de datos inválidos**: limpieza de registros incompletos.
- **Clasificación por comuna y tipo de incidente**: agrupación lógica de los datos para detectar zonas críticas o eventos frecuentes.

- **Análisis temporal**: agrupación por fecha para detectar picos y patrones.

Los resultados fueron exportados como archivos CSV y almacenados en carpetas locales accesibles desde el host. Esto permite visualización posterior o integración con otras herramientas analíticas.

3 Diseño de Pruebas

3.1 Parte 1 - Sistema de Caché

Para evaluar el sistema se probaron automáticamente múltiples combinaciones de configuración:

- **Tamaño de caché**: 50, 200, 500, 1500
- **Cantidad de consultas**: 500, 1000, 3000, 5000
- **Política de caché**: LRU, LFU
- **Distribución**: Poisson ($\lambda = 1$), Binomial ($p = 0.5$)

Se registraron métricas como tasa de acierto (*hit rate*). Los resultados fueron graficados automáticamente y guardados localmente.

3.2 Parte 2 - Incorporacion Apache Pig

Además de las evaluaciones de rendimiento, ahora para esta parte se incorporaron pruebas sobre la estructura y calidad de los datos utilizando Apache Pig. Se verificó la capacidad del sistema para agrupar correctamente los eventos por comuna, tipo y fecha, validando así la integridad del pipeline desde la simulación hasta el análisis distribuido.

4 Resultados y Gráficos

4.1 Parte 1 - resultados gráficos sobre memoria caché

A continuación se presentan los resultados gráficos de la tasa de aciertos según configuración de las distribuciones Poisson y Binomial, junto con los parámetros de número de memoria caché y número de consultas:

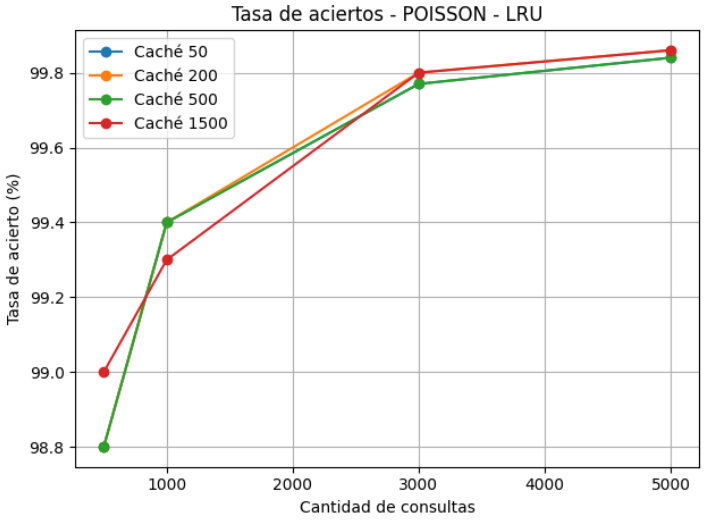

Figure 1: Tasa de aciertos - POISSON - LRU

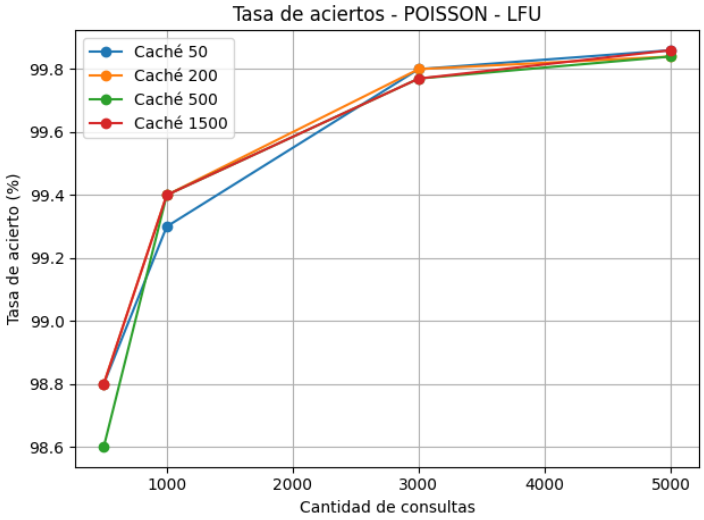

Figure 2: Tasa de aciertos - POISSON - LFU

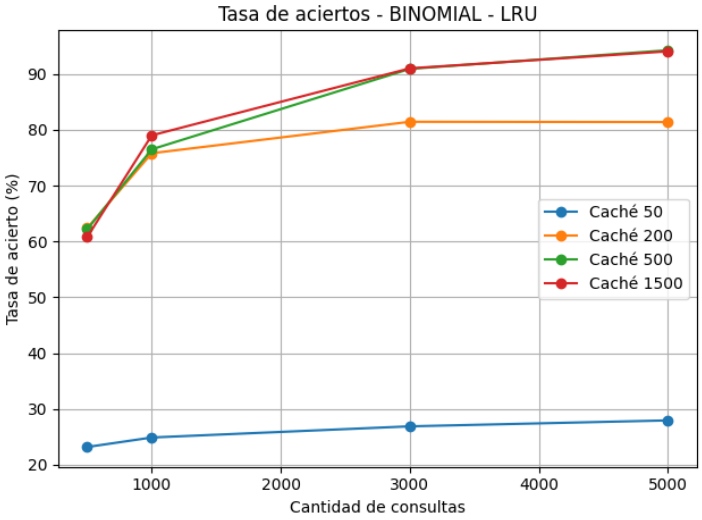

Figure 3: Tasa de aciertos - BINOMIAL - LRU

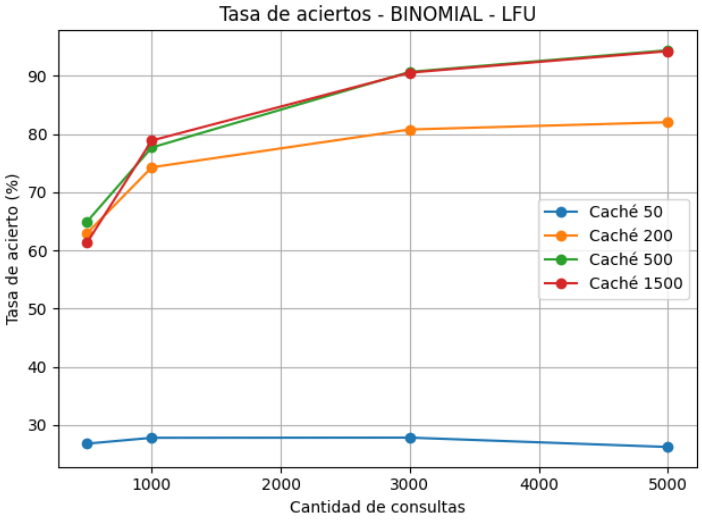

Figure 4: Tasa de aciertos - BINOMIAL - LFU

4.2 Parte 2 - Resultados del procesamiento Pig

Los resultados obtenidos a partir del script `procesar_eventos.pig` muestran:

- Las comunas con mayor número de incidentes, como `Peñalolen`, `Providencia` y `Cerro Navia`.
- Los tipos de incidentes más reportados fueron `Semáforo apagado`, `Policía` y `Desvío`.
- Se identificaron ciertas fechas (en meses) con mayor volumen de incidentes, revelando patrones temporales clave.

Cada uno de estos resultados fue exportado como archivo .csv desde Apache Pig y pudo ser utilizado para análisis posterior, por ejemplo: como las visualizaciones gráficas que se muestran a continuación:

Se generaron gráficos a partir de los resultados exportados por Apache Pig. Estos gráficos permiten observar tendencias geográficas, tipológicas y temporales sobre los eventos de tráfico simulados.

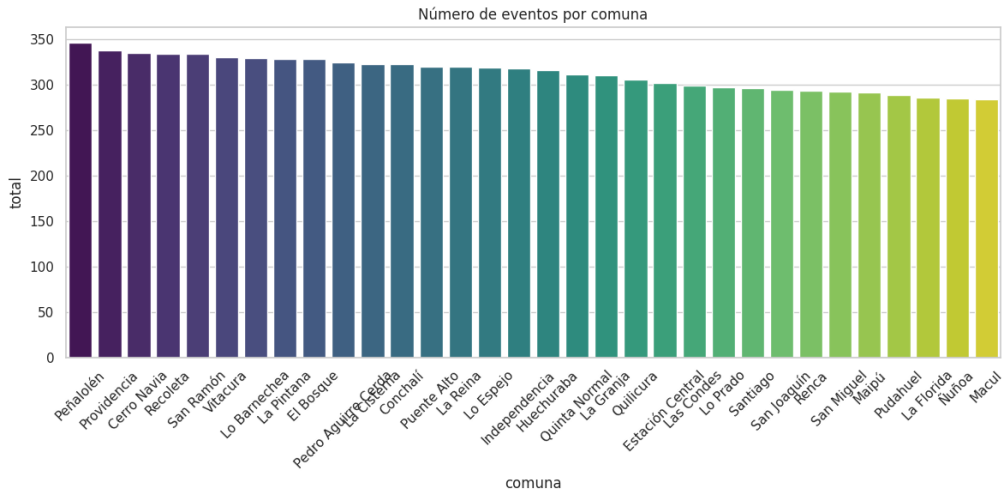

Figure 5: Número de incidentes por comuna

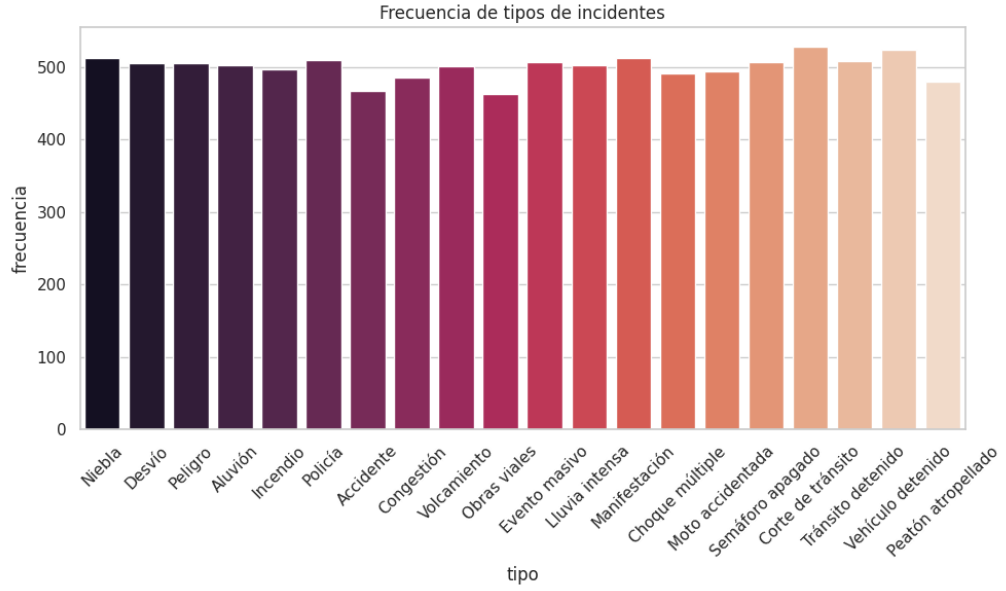

Figure 6: Frecuencia de tipos de incidentes

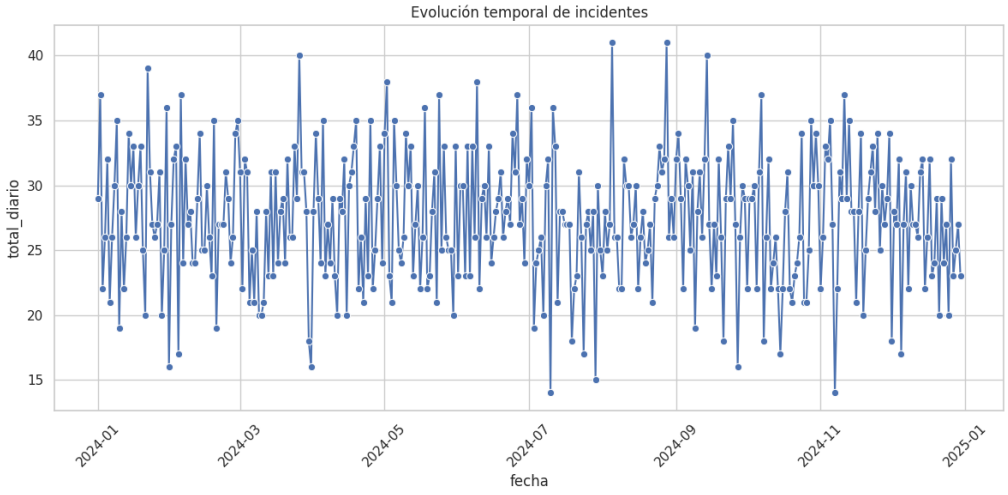

Figure 7: Evolución diaria de eventos

5 Análisis y Discusión - Parte 1 - Sistema de Caché

5.1 Generador de Tráfico

El generador de tráfico simula dos tipos de patrones de acceso mediante distribuciones estadísticas:

- **Distribución Poisson** con $\lambda = 1$: genera una alta concentración de accesos en los primeros identificadores, emulando escenarios con eventos muy repetidos (por ejemplo, congestión constante en una intersección).
- **Distribución Binomial** con $p = 0.5$: distribuye las consultas de manera más uniforme entre los eventos, modelando tráfico más equilibrado y menos repetitivo.

Estas elecciones permiten comparar cómo responden las políticas de reemplazo de caché ante distintos patrones de carga.

¿Por qué se utilizaron estas distribuciones?

5.1.1 Distribución Poisson

La distribución Poisson es ampliamente usada en teoría de colas, análisis de tráfico y sistemas distribuidos porque:

- Modela la ocurrencia de eventos en un intervalo fijo de tiempo o espacio.
- Supone que los eventos ocurren de forma independiente y con una tasa promedio constante.

Eventos reales que puede simular:

- Accidentes frecuentes en ciertas intersecciones.
- Congestión habitual en horas punta.
- Reportes reiterativos de usuarios en zonas de alta actividad.

En el sistema, Poisson genera muchas consultas hacia los mismos eventos, lo que **favorece a políticas como LFU**, que premian la frecuencia de acceso.

5.1.2 Distribución Binomial

La distribución Binomial es útil cuando:

- Hay una probabilidad fija de que un evento ocurra (éxito o fracaso).
- Se cuenta con una cantidad determinada de intentos o usuarios.

Eventos reales que puede simular:

- Probabilidad de que un usuario reporte un control policial en cierto punto.
- Consultas distribuidas homogéneamente en distintas zonas urbanas.
- Escenarios donde no hay puntos críticos, y el tráfico está más repartido.

En este caso, la distribución Binomial produce una dispersión más uniforme, lo que **beneficia a LRU**, que da prioridad al acceso más reciente, sin importar la frecuencia.

5.1.3 Resumen comparativo

Característica	**Poisson**	**Binomial**
Simula	Repetición alta	Distribución equilibrada
Favorece	LFU	LRU
Representa	Eventos frecuentes	Eventos variados
Ejemplo real	Congestión puntual	Usuarios dispersos
Parámetro clave	λ (tasa)	p (probabilidad)

5.2 Almacenamiento

Se utilizó MongoDB por las siguientes razones:

- Manejo eficiente de documentos JSON, lo que simplifica la simulación de eventos.
- Rápida inserción y consulta de documentos.
- Excelente integración con Python mediante `pymongo`.

A pesar de sus ventajas, se debe considerar la necesidad de administración de índices para escalabilidad y el uso de recursos en entornos de producción.

5.3 Métricas

La métrica utilizada fue:

- **Tasa de aciertos**: mide cuántas consultas fueron resueltas directamente desde caché.

Esta métrica es clave en sistemas distribuidos con acceso a datos frecuentes, ya que permite evaluar la eficiencia del mecanismo de caché. Una alta tasa de aciertos implica menos consultas directas a la base de datos, lo que se traduce en menor latencia, menor carga sobre el servidor y un mejor rendimiento general del sistema. Al comparar diferentes políticas de reemplazo (como LRU y LFU), esta métrica permite identificar cuál se adapta mejor al patrón de acceso del sistema simulado.

Análisis de los resultados obtenidos:

- En la distribución **Poisson**, incluso con caché pequeño (50), la tasa de aciertos superó el 90% con LFU y cerca del 80% con LRU.
- A medida que crece el tamaño del caché, ambos algoritmos alcanzan una tasa de aciertos mayor al 99%.
- En la distribución **Binomial**, con caché 50, las tasas de aciertos fueron menores (25%-40%), especialmente con LFU, lo cual es coherente con su dependencia de accesos frecuentes.
- A partir de tamaños de caché de 1500, ambas distribuciones muestran tasas de acierto muy altas (95%-99.5%), lo que indica buen aprovechamiento de la memoria.

5.4 Pruebas de Rendimiento

Se ejecutaron pruebas en 32 combinaciones posibles de:

- Políticas de caché: LRU y LFU
- Distribuciones: Poisson ($\lambda = 1$) y Binomial ($p = 0.5$)
- Caché: 50, 200, 500, 1500
- Consultas: 500, 1000, 3000, 5000

Conclusiones clave:

- **LFU** es más eficiente en contextos repetitivos, como el generado por Poisson.
- **LRU** se comporta mejor en escenarios donde las consultas se distribuyen más equitativamente.
- A medida que aumenta el tamaño del caché, ambas políticas convergen a tasas de aciertos altas, validando la correcta implementación del sistema.
- Los gráficos generados reflejan patrones esperados según la teoría, confirmando la validez de las simulaciones.

6 Análisis y Discusión - Parte 2 - Apache Pig

La integración de Apache Pig permitió transformar y explorar los datos simulados mediante operaciones de tipo MapReduce. A través del script desarrollado, fue posible:

- Eliminar registros incompletos o inválidos, asegurando consistencia.
- Agrupar y contar eventos por **comuna**, detectando zonas críticas.
- Clasificar incidentes por **tipo**, identificando los más frecuentes.
- Analizar la **evolución temporal** de los eventos, evidenciando comportamiento a lo largo de los meses específicos.

Esta etapa añadió valor al sistema al demostrar cómo los datos recolectados pueden convertirse en información útil para usuarios reales, como unidades de control de tránsito o municipios.

7 Conclusión

Se diseñó e implementó exitosamente un sistema distribuido que simula eventos de tráfico, los inserta en una base de datos y los consulta utilizando políticas de caché optimizadas. El sistema fue contenerizado, automatizado y evaluado mediante múltiples combinaciones de configuración. Se concluye que:

- El sistema cumple con los objetivos propuestos en cuanto a escalabilidad, modularidad y funcionalidad.
- LFU y LRU tienen rendimientos distintos según la distribución de consultas, lo cual permite seleccionar la política más adecuada según el escenario.
- Las pruebas y los gráficos permiten visualizar claramente la efectividad de cada estrategia.

Adicionalmente, se integró exitosamente el motor Apache Pig al pipeline del sistema, permitiendo el procesamiento distribuido de los datos de tráfico. Esta integración facilitó la limpieza, categorización y análisis exploratorio de los eventos, logrando resultados que reflejan fielmente los patrones de tráfico simulados y que podrían ser fácilmente extendidos a entornos reales.

8 Justificación de Decisiones Técnicas

- **Apache Pig** fue elegido por su simplicidad para transformar y analizar grandes volúmenes de datos estructurados. Su sintaxis declarativa permite traducir instrucciones simples en tareas MapReduce ejecutadas eficientemente.
- **Docker** permitió contenerizar el sistema completo, incluyendo Python, MongoDB y Pig. Esto garantiza replicabilidad, fácil despliegue y consistencia entre entornos.
- **CSV como formato intermedio** facilita la interoperabilidad entre componentes, ya que puede ser leído tanto por Apache Pig como por herramientas externas de análisis y permite graficar de forma facil con diferentes herramientas.

Repositorio del Proyecto

El repositorio también incluye el nuevo script `procesar_eventos.pig` y todos los archivos resultantes del procesamiento distribuido, junto con los contenedores actualizados que permiten su ejecución automatizada.

https://github.com/Kevin-css/tarea2_sd