Facultad de Química e Ingeniería del Rosario

Pontificia Universidad Católica Argentina

Cátedra de Seminario de Sistemas

Implementación de bases de datos NoSQL en el monitoreo de condiciones de Higiene y Seguridad del ambiente en entornos industriales

Presentado en cumplimiento parcial de los requisitos para la obtención del título de

LICENCIADO EN SISTEMAS Y COMPUTACIÓN

Del Fiorentino, Nicolás

Fiorenza, Agustín

Rosario, Diciembre, 2017

Índice

Introducción

Durante el cursado de la carrera, ha surgido varias veces la temática de las bases de datos en todas las áreas de sistemas. Esto nos ha despertado interés en las mismas, y luego de investigar, surgió en reiteradas oportunidades la temática de las nuevas bases de datos no relacionales con distintas variantes y aplicadas en ámbitos variados, desde sistemas convencionales de gestión hasta sistemas de reconocimiento facial.

Otra temática que estaba surgiendo era IoT. Estas aplicaciones, sensan grandes cantidades de datos que deben ser persistidos para su posterior procesamiento, por lo que pensamos que podríamos investigar conjuntamente estas dos tecnologías emergentes.

En base al trabajo de investigación titulado "*Análisis y diseño de una red sensores con tecnología inalámbrica para el monitoreo de condiciones de Higiene y Seguridad del ambiente en entornos industriales"*, realizado por la facultad, se nos presentó la oportunidad de colaborar junto a ellos investigando la posibilidad de persistir los datos en una base de datos NoSQL y que la misma logre los objetivos planteados para poder implementar el sistema exitosamente.

En esta tesina se presenta una actualización teórica referida a las bases de datos NoSQL, y se analizan dos tipos en particular de estas bases de datos: Redis, que es una base de datos clave-valor, y Neo4J, base de datos orientada a grafos.

Estudio del problema

Presentación y planteo del problema

Nuestra problemática nace en el marco de la investigación actualmente llevada a cabo por la Universidad Católica Argentina sede Rosario (2017).

El problema que intentamos resolver implica la toma de datos ambientales de sensores esparcidos dentro de la superficie de una planta industrial. Se propone esta recolección de datos que conforman las condiciones de higiene y seguridad ambientales de un puesto de trabajo, para dar soporte a la inspección anual de un auditor externo, por lo que no se considera esta recolección como crítica dado que no controlan ningún dispositivo. No hay necesidad de efectuar un monitoreo minucioso ya que dichos datos servirán para conformar una base de datos a los efectos de verificar el cumplimiento de las condiciones necesarias para el desarrollo de un trabajo en un período de tiempo.

Según la definición de ISA100 podemos ubicar nuestro problema como perteneciente a la clase de aplicación de monitoreo de condiciones no críticas. Esto nos lleva a que para solucionar el problema podemos utilizar protocolos y dispositivos que no necesariamente están diseñados para entornos industriales. Por lo tanto se utilizan en este proyecto protocolos y dispositivos generalmente usados para el sensado de datos ambientales.

Desde el punto de vista de la energía tampoco tendremos ninguna dificultad, dado que por tratarse de una planta industrial, dispondremos de fácil acceso a energía eléctrica. Otra cuestión que debería considerarse es el tema de la interferencia que los equipamientos industriales podrían hacer sobre los dispositivos inalámbricos, este tema se puede solucionar variando la ubicación de los dispositivos y no será abordado en este trabajo. (Rodríguez,E., Deco, C. Burzacca, L., Pettinari, M., Costa, S., Bender, C., 2015, P.1)

Ley N° 19587

Dentro de sus normas generales se destaca para este trabajo el artículo 4, que comprende las normas técnicas y medidas sanitarias, precautorias, de tutela o de cualquier otro índole que tenga por objeto:

a) Proteger la vida, preservar y mantener la integridad psicofísica de los trabajadores;

b) Prevenir, reducir, eliminar o aislar los riesgos de los distintos centros o puestos de trabajo;

c) Estimular y desarrollar una actitud positiva respecto de la prevención de los accidentes o enfermedades que puedan derivarse de la actividad laboral.

También se destaca del artículo 6 de esta ley, con las reglamentaciones de las condiciones de higiene y seguridad de los ambientes de trabajo; en particular el inciso b) factores físicos, referido a: cubaje, ventilación, temperatura, carga térmica, presión, humedad, iluminación, ruido, vibraciones y radiaciones ionizantes.

Capítulo 8 - Temperatura

Art. 60 - Definiciones:

- **Carga térmica ambiental:** es el calor intercambiado entre el hombre y el ambiente.
- **Carga térmica:** es la suma de carga térmica ambiental y el calor generado en los procesos metabólicos.
- **Condiciones higrotérmicas:** son las determinadas por la temperatura, humedad, velocidad de aire y radiación térmica. Evaluación de las condiciones higrotérmicas.

Instrumental a emplear

Los aparatos que se enumeran a continuación constituyen un conjunto mínimo para la evaluación de la carga térmica.

El globotermómetro mide la temperatura del globo y consiste en una esfera hueca de cobre, pintada de color negro mate, con un termómetro o termocupla inserto en ella. Se verificará la lectura del mismo cada 5 minutos, leyendo su graduación a partir de los primeros 20 minutos hasta obtener una lectura constante.

El termómetro de bulbo húmedo natural medirá la temperatura y consiste en un termómetro cuyo bulbo está recubierto por un tejido de algodón. Este deberá mojarse con agua destilada durante no menos de media hora antes de efectuar la lectura y estará sumergido en un recipiente conteniendo agua destilada.

La estimación del calor metabólico se realizará por medio de tablas según la posición en el trabajo y el grado de actividad. Se considerará el calor metabólico (M) como la sumatoria del metabolismo basal (MB), y las adiciones derivadas de la posición (MÍ) y del trabajo (MII) por lo que

$M = MB + MI + MII$ en donde:

- Metabolismo Basal (MB): MB = 70W
- Adición derivada de la posición (MI)
    - Acostado o Sentado: 21
    - De pie: 42
    - Caminando: 140
    - Subiendo pendiente: 210
- Adición derivada del tipo de trabajo (MII): En W
    - Trabajo manual ligero: 28
    - Trabajo manual pesado: 63
    - Trabajo con un brazo: ligero 70
    - Trabajo con un brazo: pesado 126
    - Trabajo con ambos brazos: ligero 105
    - Trabajo con ambos brazos: pesado 175
    - Trabajo con el cuerpo: ligero 210
    - Trabajo con el cuerpo: moderado 350
    - Trabajo con el cuerpo: pesado 490

- Trabajo con el cuerpo: muy pesado 630
- Coef. = 1,563 para pasar de Kcal/h a Watt.

A efectos de evaluar la exposición de los trabajadores sometidos a carga térmica, se calculará el Índice de Temperatura Globo Bulbo Húmedo (TGBH). Este cálculo partirá de las siguientes fórmulas:

Para lugares interiores o exteriores sin carga solar:

$$TGBH = 0,7\,TBH + 0,3\,TG$$

Para lugares exteriores con carga solar:

$$TGBH = 0,7\,TBH + 0,2\,TG + 0,1\,TBS$$

Donde:

- TGBH: índice de temperatura globo bulbo húmedo.
- TBH: temperatura del bulbo húmedo natural.
- TBS: temperatura del bulbo seco.
- TG: temperatura del globo.

Límites permisibles para la carga térmica. Valores dados en ℃ - TGBH.

	Tipo de trabajo		
Régimen de trabajo y descanso	Liviano (menos de 230 W)	Moderado (230 - 400 W)	Pesado (más de 400 W)
Trabajo continuo	30	26.7	25
75 % trabajo y 25 % - descanso, cada hora	30.6	28	25.9
50 % trabajo y 50 % - descanso, cada hora	31.4	29.4	27.9
25 % trabajo y 75 % - descanso, cada hora	32.2	31.1	30

Trabajo continuo: Ocho horas diarias (48 horas semanales)

Según la investigación se utilizará el sensor de Temperatura y Humedad DHT22. Es un sensor que mide la temperatura desde -40 a 80 ℃, por lo que usaremos esta unidad de medida en vez TGBH.

Capítulo 12 - Iluminación

Art. 71 - La iluminación en los lugares de trabajo deberá cumplimentar lo siguiente:

1. La composición espectral de la luz deberá ser adecuada a la tarea a realizar, de modo que permita observar o reproducir los colores en la medida que sea necesario.
2. El efecto estroboscópico será evitado.
3. La iluminación será adecuada a la tarea a efectuar, teniendo en cuenta el mínimo tamaño a percibir, la reflexión de los elementos, el contraste y el movimiento.
4. Las fuentes de iluminación no deberán producir deslumbramiento, directo o reflejado, para lo que se distribuirán y orientarán convenientemente las luminarias y superficies reflectantes existentes en el local.
5. La uniformidad de la iluminación, así como las sombras y contrastes, serán adecuados a la tarea que se realice

Art. 76 - En todo establecimiento donde se realicen tareas en horarios nocturnos o que cuenten con lugares de trabajo que no reciben luz natural en horarios diurnos deberá instalarse un sistema de iluminación de emergencia. Este sistema suministrará una iluminancia no menor de 30 luxes a 80 cm del suelo y se pondrá en servicio en el momento de corte de energía eléctrica, facilitando la evacuación del personal en caso necesario e iluminando los lugares de riesgo.

El anexo IX establece "*La intensidad mínima de iluminación, medida sobre el plano de trabajo, ya sea éste horizontal, vertical u oblicuo, está establecida en la tabla 1, de acuerdo con la dificultad de la tarea visual y en la tabla 2, de acuerdo con*

el destino del local. Los valores indicados en la tabla 1, se usarán para estimar los requeridos para tareas que no han sido incluidas en la tabla 2."

Para asegurar una uniformidad razonable en la iluminancia de un local, se exigirá una relación no menor de 0,5 entre sus valores mínimos y medio.

$$E\ mínima\ >\ E\ media/2$$

$$E\ =\ Exigencia$$

La iluminancia media se determinará efectuando la media aritmética de la iluminancia general considerada en todo el edificio, y la mínima será el menor valor de iluminancia en las superficies de trabajo o en un plano horizontal a 0,80 m. del suelo.

Según la investigación se utilizará el sensor de Luminosidad TSL2561. Es un sensor de luz digital avanzado, que permite una medición exacta de iluminancia en diversas condiciones de iluminación en un rango desde 0.1 a 40000 lux.

Tabla 1 - Intensidad Media de Iluminación para Diversas Clases de Tarea Visual (Basada en Norma IRAM-AADL J 20-06)

Clase de tarea visual	Iluminación sobre plano de trabajo (lux)	Ejemplo de tareas visuales
Visión ocasional solamente	100	Para permitir movimientos seguros por ej. en lugares de poco tránsito: Sala de calderas, depósito de materiales voluminosos y otros.
Tareas intermitentes ordinarias y fáciles, con contrastes fuertes.	100 a 300	Trabajos simples, intermitentes y mecánicos, inspección general y contado de partes de stock, colocación de maquinaria pesada.
Tarea moderadamente crítica y prolongadas, con	300 a 750	Trabajos medianos, mecánicos y manuales, inspección y montaje; trabajos comunes de oficina, tales como: lectura, escritura y archivo.

detalles medianos		
Tareas severas y prolongadas y de poco contraste	750 a 1500	Trabajos finos, mecánicos y manuales, montajes e inspección; pintura extrafina, sopleteado, costura de ropa oscura.
Tareas muy severas y prolongadas, con detalles minuciosos o muy poco contraste	1500 a 3000	Montaje e inspección de mecanismos delicados, fabricación de herramientas y matrices; inspección con calibrador, trabajo de molienda fina.
Tareas excepcionales, difíciles o importantes	3000 5000 a 10000	Trabajo fino de relojería y reparación. Casos especiales, como por ejemplo: iluminación del campo operatorio en una sala de cirugía.

Tabla 2[1] - Intensidad Mínima de Iluminación (Basada en Norma IRAM-AADL J 20-06)

Capítulo 13 - Acustica

Art. 85.- En todos los establecimientos, ningún trabajador podrá estar expuesto a una dosis de nivel sonoro continuo equivalente superior a la establecida en el Anexo V.

El anexo V indica sobre la dosis máxima admisible "*Ningún trabajador podrá estar expuesto a una dosis superior a 90 dB(A) de Nivel Sonoro Continuo Equivalente, para una jornada de 8 h. y 48 h. semanales. Por encima de 115 dB(A) no se permitirá ninguna exposición sin protección individual ininterrumpida mientras dure la agresión sonora. Asimismo en niveles mayores de 135 dB(A) no se permitirá el trabajo ni aún con el uso obligatorio de protectores individuales.*".

Según la investigación, para medir estos niveles se utilizará el sensor de sonido analógico DFR0034. Este sensor mide el sonido del ambiente y digitalmente

[1] Link al anexo donde se encuentra la tabla http://servicios.infoleg.gob.ar/infolegInternet/anexos/30000-34999/32030/dto351-1979-anexo4.htm

registra los valores en decibelios. La ley indica que la medición se efectuará con el micrófono ubicado a la altura del oído del trabajador.

Cálculo del nivel sonoro de ruidos no impulsivos

Sí los ruidos son continuos y sus variaciones nos sobrepasan los +- 5 dB, se promediarán los valores obtenidos en una jornada típica de trabajo. Sí los ruidos son discontinuos o sus variaciones sobrepasan los +- 5 dB, se hará una medición estadística, clasificando los niveles en rangos de 5 dB y computando el tiempo de exposición a cada nivel. Con los valores obtenidos se computará el nivel sonoro continuo equivalente (N.S.C.E.) en base semanal.

Cálculo del nivel sonoro de ruidos de impacto

Se considerarán ruidos de impacto a aquellos que tienen un crecimiento casi instantáneo, una frecuencia de repetición menor de 10 por segundo y un decrecimiento exponencial. Cuando la frecuencia de repetición de los ruidos de impacto sea superior a los 10 por segundo, deberán considerarse como ruidos continuos.

Cálculo del nivel sonoro de ruidos impulsivos

Se considerarán ruidos impulsivos aquellos que tienen un crecimiento casi instantáneo y una duración menor de 50 milisegundos.

Tabla de exposición diaria

EXPOSICIÓN DIARIA		NIVEL MAXIMO PERMISIBLE
HORAS	MINUTOS	dB (A)
8	-	90
7	-	90.5
6	-	91
5	-	92
4	-	93
3	-	94

2	-	96
1	-	99
-	30	102
-	15	105
-	1	115

Significado NoSQL

Definición

NoSQL es un término genérico para referirse a cualquier base de datos que no sigue el paradigma tradicional de las bases de datos relacionales. Específicamente, los datos se persisten de forma no relacional y no usa SQL como lenguaje de consulta. Se utiliza para referirse a las bases de datos que intentan resolver los problemas de escalabilidad y disponibilidad frente a la de la atomicidad y consistencia.

Marco histórico

El término NoSQL surge por primera vez a finales de la década del 90, como nombre de una base de datos relacional open source llamada 'Strozzi NoQSL'. Encabezado por Carlo Strozzi, esta base de datos almacena sus tablas en archivos ASCII, donde cada línea representa una tupla con campos separados mediante tabs.

El nombre proviene del hecho que la base de datos no utiliza SQL (Structured Query Language) como lenguaje de consulta. En cambio, los datos son manipulados mediante scripts en línea de comandos que se podían combinar con pipelines de UNIX. Pipeline es una secuencia de operaciones donde la salida de una operación es la entrada de la siguiente.

El término NoSQL como lo conocemos hoy en día resurge en una reunión el 11 de junio de 2009 en San Francisco organizada por Johan Oskarsson, un desarrollador de software oriundo de Londres. La aparición de BigTable y Dynamo inspiró a un puñado de proyectos alternativos que estaban experimentando con opciones alternativas de persistencia de datos y estas discusiones eran un tópico importante en las conferencias de desarrollo de software.

Johan estaba interesado en investigar estas nuevas bases de datos durante su estadía en San Francisco para la conferencia de Hadoop, por lo que organizó una reunión para que todos los interesados puedan asistir y presentar su trabajo a quien estuviese interesado.

Johan quería un nombre para la reunión, un nombre que fuera bueno para un hashtag de Twitter: corto, memorable y sin demasiada información en Google para que pudiera ser encontrado fácilmente. Consultó en foros de Cassandra y obtuvo algunos, eligiendo la opción “NoSQL” . Mientras tenía la desventaja de ser negativo y no describir realmente estos sistema, seguía los criterios de un hashtag.

El término NoSQL tuvo amplia aceptación para describir estas nuevas tecnologías, pero nunca fue un término que describiera de forma clara estos sistemas. Las tecnologías expuestas en la reunión fueron Voldemort, Cassandra, Dynomite, HBase, Hypertable, CouchDB y MongoDB. Aunque el término nunca ha sido confinado a este septeto inicial, no hay una definición generalmente aceptada, o una autoridad que provea uno, entonces el término que prevalece para hablar sobre base de datos que tienen ciertas caracteristicas en comun es el de NoSQL.

Para empezar, las bases de datos NoSQL no utilizan SQL. Algunas tienen lenguajes de consulta similares a SQL para facilitar su aprendizaje, como por ejemplo Cassandra CQL. Aunque ninguna ha desarrollado un lenguaje que implemente los estándares de SQL.

Otra característica importante es que generalmente son proyectos de código libre o abierto.

La mayoría de los proyectos NoSQL están orientados a las necesidades de correr en clusters, esto tiene consecuencias en el modelado de datos como también con su enfoque de consistencia. Las bases de datos relacionales usan transacciones ACID para asegurar la consistencia en toda la base de datos. Esto choca con la idea de un ambiente orientado a clusters, por lo que las bases de datos NoSQL ofrecen una gama de opciones para asegurar la consistencia y los datos distribuidos.

Las bases de datos NoSQL están basadas en las necesidades del siglo XXI, sobre todo en un enfoque web o en la nube. Las mismas operan sin esquemas definidos, permitiendo agregar campos a los registros sin tener que predefinir los cambios en la estructura. Esta característica es útil cuando se debe trabajar con datos informales o personalizados que fuerzan a las bases de datos relacionales a usar nombres que son difíciles de procesar y entender.

Las características mencionadas son comunes a todas las bases de datos NoSQL.

El teorema de Brewer o teorema CAP[2]

Eric Brewer menciona que es imposible para un sistema distribuido proveer consistencia, disponibilidad y tolerancia a la partición de forma simultánea. Esto fué lo que desencadenó el teorema de CAP.

- Consistencia (Consistency): Implica que la información permanece coherente y consistente después de cualquier operación sobre los datos.
- Disponibilidad (Availability) : Significa que toda la información del sistema de almacenamiento de datos distribuido está siempre disponible.
- Tolerancia a las Particiones (Partition tolerance): Significa que el sistema de almacenamiento distribuido puede seguir funcionando si la conexión entre algunos de los nodos sea interrumpida o falle.

[2] Brewer, Eric A. (2000) : *Towards Robust Distributed Systems*. Portland, Oregon.

Según Eric Brewer no es posible cumplir todas las dimensiones a la vez, hay que elegir dos de ellas. Esto genera una clasificación de los sistemas distribuidos.

- **Consistencia y Disponibilidad**: Son sistemas que si la conexión entre los nodos es inestable o falla, el sistema no puede trabajar.

Se puede aplicar a Bases de datos de un sólo sitio, Clusters de bases de datos y el sistema de archivo xFS file system. Estas bases de datos poseen commit de 2 fases o protocolos de validación de cache.

- **Consistencia y Tolerancia a las Particiones**: Son sistemas en los cuales si hay una contingencia parte de la información no estará disponible, pero la información disponible será consistente. Ej. MongoDB y Paxos.

Una forma de utilizar esta combinación es con bases de datos distribuidas, lockeo distribuido y protocolo Majority de lockeo. Estas bases de datos poseen una estrategia de bloqueo pesimista.

- **Disponibilidad y Tolerancia a las Particiones**: Durante una contingencia entre los nodos la información estará disponible, pero puede que no sea consistente. Ej. Cassandra y CouchDB.

Esto se aplica a almacenamiento de caché web y servidores DNS. Además poseen las siguientes características: Expiraciones, resolución de conflictos y es optimista.

Según Christof Strauch cuando la consistencia y la tolerancia a las particiones son los dos focos de interés, las propiedades BASE son las que mejor se aplican. Mientras que si la Disponibilidad y la tolerancia a las particiones son los focos, las propiedades ACID son las que mejor se aplican.[3]

ACID vs BASE

Las bases de datos relaciones se enfocan en transacciones ACID:

[3] Strauch Christof (2011): *NoSQL Databases. (pp 30-31).* Stuttgart University. Stuttgart, Alemania.

- **ATOMICIDAD**: Todas las operaciones de una transacción deben tener éxito para que se complete la transacción.
- **CONSISTENCIA**: Una transacción no puede finalizar en un estado inconsistente.
- **AISLAMIENTO**: Una transacción no puede interferir a otra transacción.
- **DURACIÓN**: Una transacción completa persiste aunque la aplicación se reinicie.

Las bases de datos NoSQL se enfocan en lo que Eric Brewer denomina **BASE**:

- **BASIC AVAILABILITY** (DISPONIBILIDAD BÁSICA): Cada solicitud debe recibir una respuesta, exitosa o errónea.
- **SOFT STATE** (ESTADO FLEXIBLE): El estado del sistema va cambiando en función del tiempo.
- **EVENTUAL CONSISTENCY** (CONSISTENCIA EVENTUAL): La base de datos podría estar momentáneamente inconsistente pero va a volver a ser consistente finalmente.

En su charla “Patrones de Diseño para Bases de Datos no Distribuidas” Todd Lipcon realiza una diferenciación entre Consistencia Estricta y Consistencia Eventual.[4]

Consistencia Estricta: Significa que todas las operaciones de lectura devuelven datos de la última operación de escritura, sin importar en qué réplica fue ejecutada la operación. Esto implica que, o las operaciones de escritura y lectura fueron hechas en el mismo nodo, o que algún protocolo de transacciones aseguró la consistencia.

Consistencia Eventual: Significa que los lectores van a ver las escrituras a medida que pasa el tiempo. Esto significa que los clientes pueden encontrarse frente a un

[4] Lipcon, Todd (2009): *Diseño de Patrones para Bases de Datos No Relacionales.*

estado de inconsistencia a medida que las actualizaciones de las escrituras están en progreso.

Ventajas frente a bases de datos relacionales

Las tres características básicas de No-SQL son: La ausencia de esquemas en los registros de datos, escalabilidad horizontal sencilla y mayor velocidad.

La ausencia de esquemas en los registros de datos significa que los datos no tienen una definición de atributos fija, es decir que cada documento (se lo llama así a cada registro en sistemas de este tipo) puede contener una información diferente en distintos tiempos, facilitando así el poliformismo de datos bajo una misma colección de información. También se pueden almacenar estructuras de datos complejas en un sólo documento, como por ejemplo una publicación.

La escalabilidad horizontal significa la posibilidad de aumentar el rendimiento del sistema simplemente añadiendo más nodos. Los sistemas No-SQL permiten utilizar consultas del tipo Map-Reduce, que se ejecuta en distintos nodos a la vez y luego reúne los resultados.

Por otro lado, la mayor velocidad se logra ya que muchos de estos sistemas realizan operaciones directamente en memoria y sólo vuelcan los datos a disco cada cierto tiempo. Esto sumado a que los sistemas NoSQL permiten utilizar consultas del tipo Map-Reduce, que se ejecuta en distintos nodos a la vez y luego reúne los resultados, mejorando el desempeño y la velocidad de las consultas.

Modelo distribuido

El principal interés de una base de datos NoSQL es su habilidad de correr base de datos en cluster de servidores. Cuando el volúmen de datos se incrementa, se vuelve más costoso comprar un servidor con mayores prestaciones para correr la base de datos. Una opción es escalar horizontalmente, correr la base de datos en un cluster de servidores, que es más económico pero agrega complejidad.

Sharding

Es una técnica en donde distintas partes de la información residen en diferentes clusters.

En el caso ideal, se balancea la carga así cada usuario se comunica con un sólo nodo, entonces sí por ejemplo tenemos 10 nodos cada uno maneja el 10% de la carga y el usuario recibe una rápida respuesta.

Hay muchos factores que se tienen en cuenta a la hora de alojar la información en un servidor para mejorar la perfomance. Sí se sabe que la mayoría de datos van a ser requeridos en una zona geográfica en particular, se puede alojar la información en un datacenter de la zona. También se debe tener en cuenta de alojar juntos los datos agregados, esto mejora la velocidad de lectura, escritura y procesamiento de los datos.

Muchos motores de base de datos NoSQL ofrecen auto-sharding, donde las bases de datos toman la responsabilidad de ubicar los datos en los distintos nodos y asegurar que el acceso a los datos sean en el nodo correcto.

Replicación maestro-esclavo

En la replicación maestro-esclavo se replican los datos en múltiples nodos. El nodo maestro es la fuente autoritativa de los datos y responsable de procesar las actualizaciones de los datos. Los demás nodos son esclavos, un proceso de replicación sincroniza los esclavos con el maestro.

La replicación maestro-esclavo ayuda a escalar horizontalmente cuando se tiene una base de datos donde se hacen lectura de datos intensivas, se manejan más lecturas añadiendo más nodos esclavos asegurando que todas las lecturas sean manejadas por los nodos esclavos. Sin embargo estamos limitados por la capacidad del nodo maestro de procesar las actualizaciones y replicarlos a los esclavos. Por eso no es un buen esquema cuando se trata de una base de datos con alto tráfico, aunque un balanceo de cargas en la lectura de datos ayuda a manejar mejor la escritura en el nodo maestro.

La segunda ventaja es la resistencia a la hora de leer los datos, si el nodo maestro falla los nodos esclavos pueden seguir manejando la lectura de datos, aunque no permite la escritura y sincronización de datos, hasta que el nodo maestro se recupere o se apunte a nuevo maestro.

La desventaja de este modelo es la inconsistencia, diferentes cliente pueden leer diferentes valores del mismo dato en los distintos nodos esclavos si no se propaga correctamente la información. También si el nodo maestro falla, cualquier actualización que no se haya propagado a un nodo esclavo se pierde.

Otra desventaja es que el nodo maestro es el cuello de botella y el único punto de fallo.

Replicación peer-to-peer

Esta modelo ataca los problemas del modelo esclavo-maestro sacando el nodo maestro del esquema. Todos los nodos tienen el mismo peso, aceptan escrituras y la pérdida de cualquier nodo no afecta el acceso a los datos.

La mayor complejidad de este modelo es la consistencia, cuando múltiples usuarios escriben el mismo registro, se corre con el riesgo de generar un conflicto de escritura. Este problema se mitiga, permitiendo a un solo usuario escribir un registro, los demás nodos se coordinan para asegurar la consistencia.

Combinar sharding y replicación

La replicación y el sharding pueden ser combinadas.

Si usamos maestros-esclavo junto a sharding, significa que tenemos múltiples nodos maestros pero cada dato sólo tiene un nodo maestro.

La combinación peer-to-peer junto a sharding es muy común en familias de base de datos NoSQL orientadas a columnas. En entornos como estos tendremos n nodos con datos fragmentados en cada nodo.

Clasificación de bases de datos NoSQL

La clasificación es según el modelo de datos representado.

Documentos

- MongoDB: www.mongodb.com
- CouchDB: couchdb.apache.org
- RavenDB: www.ravendb.net

Clave-Valor

- Redis: www.redis.io
- Couchbase: www.couchbase.com
- Voldemort: www.project-voldemort.com
- MemcacheDB: www.memcachedb.org

XML (Extensible Markup Language)

- BaseX: www.basex.org
- eXist: www.exist-db.org

Columnas

- BigTable: cloud.google.com/bigtable/

- Hadoop: www.hadoop.apache.org
- Cassandra: cassandra.apache.org
- SimpleDB: aws.amazon.com/simpledb

Grafos

- Neo4j: www.neo4j.com
- FlockDB: www.github.com/twitter-archive/flockdb
- InfiniteGraph: www.objectivity.com/products/infinitegraph/

A partir de esta clasificación, se estudiarán bases de datos orientadas a documentos, clave-valor y columnas para determinar cuál puede ser la más factible para el proyecto.

Bases de datos a estudiar

Clave-Valor

Introducción

Una base de datos clave-valor es una tabla hash donde el acceso a los datos es mediante una clave primaria. Se la puede comparar a una tabla relacional tradicional de dos columnas, clave y valor, donde clave es el identificador único y valor es un campo tipo String. El campo valor sólo almacena los datos, sin preocuparse por los detalles de los datos que guarda, es responsabilidad de la aplicación entender el dato guardado.

Desde la perspectiva de la API, es el tipo de base de datos NoSQL más simple que existe. El cliente envía el par clave-valor, si la clave ya existe el valor se reescribe y sino crea un nuevo registro. Ya que se acceden a los datos mediante clave primaria, generalmente tienen una performance excelente y escalan fácilmente.

Casos de uso adecuados

Son útiles en caso de guardar datos que no tienen una relación entre sí, o para persistir entidades únicas, como ser un perfil de usuario.

Casos de usos inadecuados

Cuando es necesario mantener relaciones entre los datos, las bases de datos clave-valor no son la mejor solución, aunque algunas proveen un mecanismo de enlace entre datos.

Sí es necesario hacer búsquedas complejas de una clave primaria en el valor, las bases de datos clave-valor no tienen buena performance, no hay forma de inspeccionar el valor en la base de datos, sino hay que hacerlo en la aplicación.

Tampoco es posible realizar operaciones en múltiples registros a la vez, es necesario hacerlo en la aplicación.

A continuación, se analiza en profundidad la base de datos Redis.

Redis

- **Consistencia**

Las operaciones son consistentes sobre sólo una clave primaria. En base de datos clave-valor distribuidas, la consistencia se resuelve de dos formas, el valor más nuevo se escribe sobre el valor actual, o ambos valores son retornados al cliente a la espera que resuelva el conflicto.

- **Transacciones**

Redis soporta transacciones mediante los comandos MULTI, EXEC, DISCARD y WATCH**.** Permite la ejecución de un grupo de comandos garantizando que todos los comandos dentro de la transacción son ejecutados secuencialmente y de forma aislada, por lo tanto ningún otro comando de otro cliente va a interferir la

ejecución de la transacción y también garantiza que todos los comandos son ejecutados o ninguno, es decir de forma atómica.

Redis no soporta comandos Rollback, argumentando que sólo falla en caso de que un comando fue ejecutado con una sintaxis incorrecta o con un tipo de dato incorrecto y no pretenden sacrificar performance por agregar esta característica.

- **Consulta de datos**

Todas las base de datos clave-valor permiten hacer búsquedas por la clave, si es necesario hacer búsquedas en el valor generalmente las base de datos de este tipo no lo soportan.

- **Tipos de datos**

Redis soporta desde un String hasta estructuras de datos complejas en el valor. Algunos de estos tipos de datos son Lists, Sets, Hashes. En la clave soporta cualquier dato binario, desde un id o String hasta el contenido de un archivo JPEG. También soporta una cadena vacía.

- **Persistencia**

Redis es un motor de bases de datos en memoria, lo que es una ventaja en performance y disponibilidad pero es volátil, por lo que Redis permite persistir los datos mediante dos mecanismos: RDB y AOF.

El mecanismo RDB hace snapshots en el tiempo del conjunto de datos en intervalos de datos especificados. Son perfectos para backups y recuperación ante fallos, ya que persiste toda la información en un sólo archivo y se puede configurar backups automáticos. Es eficiente ya que cuando hace el backup lo único que hace es una copia de la instancia de la base de datos en ese momento, no ejecuta operaciones de E/S, aunque es lento cuando debe persistir grandes conjuntos de datos.

El mecanismo AOF es más duradero y posee mejores configuraciones. Se ejecuta en segundo plano por lo tanto es eficiente en grandes conjuntos de datos.

AOF también mantiene un log de todas las operaciones ejecutadas en la base de datos.

- **Alta disponibilidad**

Redis Sentinel provee alta disponibilidad para redis. Sentinel provee un mecanismo que previene fallos sin intervención humana. También posee monitoreo de nodos, notificaciones de fallos, control de fallos automático es nodos maestros y si falla el nodo maestro Sentinel le provee la nueva dirección de un nuevo nodo Master a los demás slaves.

- **Cluster**

Redis posee Redis Cluster, provee una forma de correr Redis distribuido donde los datos son automáticamente compartidos en múltiples nodos. Tiene la habilidad de continuar la operación aunque algunos de los nodos fallen o no se puedan comunicar, aunque el clúster falla si la varios nodos Master fallan.

- **Sistemas Operativos**

Redis funciona en la mayoría de los sistemas operativos basados en POSIX como ser Linux, BSD y OSX sin ninguna dependencia externa. No tiene soporte oficial para windows pero Microsoft desarrolló y mantiene una versión de redis para Windows en 64 bits.

- **Lenguajes de programación**

Redis tiene soporte en la mayoría de los lenguajes de programación como ser Python, Java, Javascript, C, C#, C++, PHP, etc.

- **Licencia**

Redis está desarrollado bajo el concepto de código abierto y licencia BSD. BSD significa Berkeley Software Distribution, un tipo de sistema operativo unix-like. Esta licencia posee menos restricciones en comparación con otras licencias como GNU.

Documentos

Introducción

Las bases de datos orientadas a documentos guardan y recuperan documentos; que pueden tener formato XML, JSON, entre otros. Estos documentos son autodescriptivos, con una jerarquía de árbol que pueden consistir en maps, collections o valores escalares.

El esquema de los datos pueden diferir entre los documentos, aunque pertenezcan a la misma colección de datos, contrario a las bases de datos relacionales que cada tupla de la misma tabla contiene los mismos atributos. Embeber documentos dentro de otros documentos permite un fácil acceso a los datos y mejoras en performance.

Otra diferencia de las bases de datos orientadas a documentos es que no existen los atributos sin valor, si un atributo no se encuentra, es que no se considera relevante para ese documento. Este tipo de base de datos también permite a los nuevos atributos ser creados sin necesidad de definirlos o modificar los documentos ya existentes.

Casos de uso adecuados

Son muy populares en entornos de desarrollo web donde hay alta mutación de datos, necesidad de performance y facilidad de recuperar los datos.

Casos de usos inadecuados

Estos esquemas flexibles no fuerza restricciones en el esquema, por lo tanto no mantienen los datos íntegros desde la perspectiva de la base de datos. Como los datos son guardados como agregados, si el diseño de los datos mutan constantemente y se necesita que los datos estén normalizados, las bases de datos documentales pueden no funcionar.

En este tipo de base de datos NoSQL se analiza en profundidad MongoDB.

MongoDB

- **Consistencia**

La consistencia en MongoDB es configurada usando un mecanismo llamado **replica sets**. Replica set es un conjunto de instancias mongod que mantienen el mismo set de datos, utilizan el modelo distribuido maestro-esclavo.

- **Concurrencia**

Para asegurar la consistencia, utilizan bloqueos y un control de concurrencia que mide la interacción de los usuarios con el dato y previene que múltiples usuarios lo modifiquen simultáneamente.

MongoDB usa bloqueos multi-granulares, que permiten operaciones que bloqueen niveles globales, base de datos o colecciones y permiten al motor de base de datos de cada nodo a implementar su propio control de concurrencia.

Por defecto, MongoDB posee bloqueos compartidos para operaciones de lectura y bloqueos exclusivos para operaciones de escritura.

- **Transacciones**

En MongoDB, una operación de escritura es atómica en un sólo documento, incluso si la operación modifica múltiples documentos embebidos dentro de un sólo documento.

Cuando una operación de escritura modifica múltiples documentos, esta es atómica, pero la operación en conjunto no lo es y otras operaciones pueden intercalarse. Sin embargo, se puede aislar una operación de escritura con el comando $isolated.

Usando el operador $isolated, una operación de escritura que afecta múltiples documentos puede prevenir que otros procesos se intercalen una vez que se modifica el primer documento. Esto asegura que ningún cliente vea los cambios

hasta que se completen todas las operaciones de escritura o error, aunque este comando no provee una atomicidad en todas las operaciones, por lo tanto un error durante la escritura no ejecuta un rollback de todas las operaciones anteriores.

- **Consulta de datos**

Una ventaja de las base de datos documentales es que podemos hacer búsquedas de datos dentro del documento sin tener que recuperar todo el documento por su clave y después investigarlo.

MongoDB posee un lenguaje de búsqueda expresado en formato JSON, que guarda muchas similitudes con el lenguaje SQL, lo que lo posiciona como una ventaja frente a otras base de datos NoSQL.

- **Tipos de datos**

MongoDB guarda los documentos en un formato llamado BSON, binary serialization format. BSON soporta una gran cantidad de tipo de datos como valores en sus documentos, entre los que se encuentran String, Object, Array, Double, Boolean, Date, entre otros.

- **Persistencia**

MongoDB provee una variedad de motores de persistencia, tanto en memoria como en disco.

WiredTrigger es el motor de persistencia por defecto. Cuando escribe en disco, WiredTrigger escribe todos los datos en un snapshot en disco. El nuevo snapshot actúa como un checkpoint de los datos en el tiempo. El checkpoint asegura que los datos son consistentes, por lo que actúa como un mecanismo de recuperación.

Desde la versión 3.2.6, la persistencia en memoria es permitida en sistemas de 64 bits.

- **Alta disponibilidad**

MongoDB permite alta disponibilidad usando replica sets. En replica sets, dos o más nodos participan replicando los datos de forma asincrónica, en un mecanismo de replicación maestro-esclavo. Generalmente, replica sets es utilizado para redundancia de datos, automatizar recuperación ante errores, escalar lectura de datos, mantenimiento de servidores sin baja en el servicio y recuperación ante desastres.

- **Cluster**

MongoDB posee la opción de instalarse en cluster de servidores. Utiliza la técnica de replicación maestro-esclavo y sharding.

- **Sistemas Operativos**

MongoDB está desarrollado en C++ por lo que permite la instalación tanto en Linux, Debian, Ubuntu y Windows.

- **Lenguajes de programación**

MongoDB posee distintos drivers para trabajar con los lenguajes de programación más populares como pueden ser C, C++, C#, Java, NodeJS, PHP, Ruby, entre otros.

- **Licencia**

MongoDB está desarrollado bajo el concepto de código abierto y con la licencia GNU AGPL v3.0. AGPL significa licencia pública general de Affero que es una licencia copyleft derivada de GNU diseñada específicamente para asegurar la cooperación con la comunidad en el caso de software que corra en servidores de red.

Columnas

Introducción

Las bases de datos orientadas a columnas guardan datos en familias de columnas, donde cada fila tiene distintas columnas asociadas.

El origen de estas bases de datos surgen de la necesidad de las herramientas de analytics y business intelligence donde almacenar los datos en columnas permitían desarrollar aplicaciones más eficientes.

Casos de usos adecuados

Las bases de datos orientadas a columnas fueron diseñadas para dar alta disponibilidad, esto se debe a que no cuentan con nodos maestros sino que cualquier nodo acepta lecturas y escrituras.

Las estructuras son similares a las bases de datos relacionales por lo tanto es más familiar la migración.

También poseen la característica que sí una columna es leída con más frecuencia que las otras automáticamente son usadas como clave primaria para mejorar la performance.

Casos de usos inadecuados

Las bases de datos orientadas a columnas no son la mejor solución para sistemas que deban soportar transacciones ACID para operaciones de lectura y escritura. Estas bases de datos tampoco suelen soportar operaciones de agregado por lo tanto se debe realizar en el cliente.

Para este tipo de base de datos se ha elegido realizar un análisis más completo sobre Cassandra.

Cassandra

Cassandra es una de las bases de datos orientadas a columnas más populares. Cassandra escala rápida y fácilmente, ya que las operaciones de escritura se persisten en todos los nodos. El cluster no tiene nodo maestro, por lo tanto una operación de lectura o escritura pueden ser manejados por cualquier nodo del cluster.

La unidad básica de almacenamiento en Cassandra es una columna. Una columna en Cassandra consiste de un par nombre-valor donde el nombre también actúa como clave primaria. Cada columna también guarda un valor timestamp, es usado para expirar datos, resolver conflictos entre otros.

Una familia de columnas puede ser comparada con una tabla de una base de datos relacional. La única diferencia es que las filas no tienen que tener las mismas columnas.

Las familias de columnas son guardadas en keyspaces, es el equivalente de una base de datos.

- **Consistencia**

Los controles de consistencia en Cassandra permiten especificar distintos niveles de réplicas, donde cada nivel especifica la cantidad de réplicas que deben escribirse antes de retornar el mensaje de éxito al cliente.

- **Concurrencia**

Cassandra no posee transacciones ACID ni mecanismos de rollback o bloqueos, pero ofrece transacciones atómicas y aisladas con control de consistencia que ofrece al usuario la posibilidad de elegir cuan fuerte o eventual debe ser la consistencia de una transacción.

- **Transacciones**

Cassandra no soporta transacciones de forma tradicional. En Cassandra, una escritura es atómica a nivel de fila, lo que significa que escribir o insertar columnas para una fila será tratada como una sola escritura.

- **Consulta de datos**

Cassandra posee su propio lenguaje de consulta llamado CQL, Cassandra Query Language. CQL consiste en sentencia como SQL, aunque no es tan potente ya que no soporta joins o subconsultas, y las cláusulas where son más simples.

Cassandra recomienda modelar las columnas y familias de columnas optimizadas para las lecturas de datos, ya que CQL no es potente. Sí tenemos una columna que es más requerida que otras, es más eficiente usar ese valor como índice.

- **Persistencia**

Cuando se realiza una operación de escritura en Cassandra, los datos primero se escriben en un log, y después en una estructura en memoria conocida como memtable. Por último, los datos se persisten en disco en una estructura llamada SSTable.

Memtable es un caché de datos que Cassandra ordena por clave. Mientras más se usa una tabla, se necesita una tabla memtable más grande. Cassandra asigna dinámicamente la cantidad de memoria necesaria a cada memtable. Cuando memtable llega al límite, entonces encola los datos para ser escritos en memoria. Los datos que se escriben en disco también son eliminados del commit log. El commit log sólo sirve para recuperar los datos si falla la escritura en disco.

- **Alta disponibilidad**

Cassandra está diseñada para alta disponibilidad, ya que no hay nodo maestro en el cluster por lo que todos los nodos son peer to peer. La disponibilidad de un cluster puede ser incrementada reduciendo los niveles de consistencia. La disponibilidad está gobernada por una fórmula (R + W) > N, donde W es el número mínimo de nodos donde se deben escribir los datos, R es el número mínimo de

nodos que deben responder exitosamente a una lectura y N es el número mínimo de nodos participando de la replicación de datos.

- **Cluster**

Cassandra permite agrupar nodos en clusters. Utiliza una arquitectura de replicación peer to peer donde no existen los nodos maestros, lo que favorece la alta disponibilidad de los datos.

Cuando un cliente se conecta con un nodo, este nodo es el encargado de coordinar la operación. El nodo coordinador es el responsable

- **Sistemas Operativos**

Oficialmente Cassandra[5] está disponible para ser instalado bajo las distribuciones Unix, pero al estar desarrollado en Java, puede ser instalado en Windows.

- **Lenguajes de programación**

Cassandra posee un gran número de drivers[6] para utilizar, cubriendo los lenguajes más populares como Java, Python, Ruby, .NET, NodeJS, etc.

- **Licencia**

Cassandra es desarrollada por Apache Software Foundation, bajo la licencia Apache 2.0. Lo que implica que es una licencia permisiva, ya que no requiere que los trabajos derivados sean publicados bajo la misma licencia y tampoco exige la liberación del código fuente.

5 http://cassandra.apache.org/doc/latest/getting_started/installing.html

6 http://cassandra.apache.org/doc/latest/getting_started/drivers.html

Grafos

Introducción

Las bases de datos orientados a grafos permiten modelar entidades y relaciones entre estas entidades. Las entidades también se conocen como nodos, que tienen propiedades. Las relaciones se conocen como aristas y también pueden tener propiedades.

Las aristas tienen significado direccional. En cambio, los nodos están organizados por las relaciones que permiten encontrar patrones de interés entre los nodos. La organización del grafo permite que los datos sean guardados una vez y luego interpretados de diferentes maneras basados en las relaciones.

Usualmente, cuando guardamos una relación en una base de datos RDBMS, agregar otra relación implica cambios en el esquema y los datos, lo que en un motor de base de datos de grafos se simplifica, ya que no hay límite en el número o tipo de relaciones que un nodo puede tener, por lo que todas las relaciones pueden ser representadas en la misma base de datos.

Casos de usos adecuados

En aplicaciones de redes sociales las bases de datos orientadas a grafos se pueden implementar y usar muy eficientemente, donde las relaciones entre las entidades se pueden dar de varias formas distintas.

Este tipo de bases de datos permiten reconocer patrones en los datos, por lo que son adecuadas para aplicaciones tipo rule engines o árboles de decisión.

Casos de usos inadecuados

Las bases de datos orientadas a grafos no son adecuadas cuando se necesitan manipular grandes conjuntos de datos, este tipo de motores no son óptimos ya que cambiar una propiedad no es una operación sencilla.

Para este tipo de base de datos se ha elegido realizar un análisis más completo sobre Neo4j.

Neo4j

Neo4j es una de las bases de datos orientadas a grafos más populares. En esta, crear un grafo es muy sencillo ya que sólo se deben definir dos nodos y la relación entre ellos.

Las relaciones entre los nodos es el componente más importante de estas bases de datos, ya que no sólo tienen nodos de partida, de llegada y direccionalidad, sino que también tienen propiedades. Se puede agregar inteligencia y hacer consultas usando estas propiedades de las relaciones.

- **Consistencia**

Ya que las bases de datos orientadas a grafos operan con nodos conectados por relaciones, la mayoría de los motores no soportan la distribución de los nodos en diferentes servidores. Al estar los datos en un sólo servidor, los datos están siempre consistentes, especialmente en Neo4j que es ACID.

Cuando Neo4j corre en un cluster de servidores, una operación de escritura en el nodo maestro automáticamente se sincroniza con los nodos esclavos, mientras que los nodos esclavos están siempre disponibles para lectura.

Estos motores de bases de datos aseguran consistencia entre las transacciones, ya que tanto el nodo de partida como el de llegada deben existir.

- **Transacciones**

Neo4j es ACID. Antes de cambiar un nodo o agregar una relación a los nodos existentes, debemos empezar una transacción, caso contrario recibiremos una excepción. Las operaciones de lectura sí pueden ser realizadas sin una transacción.

- **Disponibilidad**

Neo4j provee alta disponibilidad ya que los datos se replican desde el nodo maestro a los nodos esclavos. Los nodos esclavos también pueden manejar operaciones de escrituras. Cuando los nodos esclavos reciben una operación de escritura, se sincroniza con el nodo maestro, y la escritura se compromete primero en el maestro y luego en el esclavo. Los otros nodos esclavos se irán actualizando progresivamente.

Cuando un servidor es el primero en conectarse, automáticamente se convierte en maestro. Cuando el maestro tiene una falla, el cluster elige un nuevo maestro de todos los disponibles, por lo que provee alta disponibilidad.

- **Consulta de datos**

Una consulta en una base de datos orientadas a grafos se denomina ‘traverse’.

Las bases de datos orientadas a grafos soportan los lenguajes de consulta Gremlin, como también Cypher. Neo4j también permite un lenguaje de consulta nativo que permite consultar sobre las propiedades de los nodos, el grafo o navegar por las relaciones.

Las propiedades de un nodo o de las relaciones pueden ser indexadas usando el servicios de índices, llamado Lucene, por lo que un nodo o una relación se pueden consultar por el valor. Lucene permite filtrar por la direccionalidad de las relaciones.

Para consultar relaciones más complejas existe el operador TRAVERSER, donde se pueden especificar la direccionalidad de las relaciones, orden, entre otros. Este operador necesita un nodo de inicio para empezar a operar.

- **Sistemas Operativos**

Neo4j está desarrollado en java por lo que se puede instalar tanto en sistemas basados en GNU/LINUX como en Windows. Sólo requiere la instalación de la JVM (máquina virtual de Java). El sistema operativo debe operar bajo arquitectura x64 bits.

- **Lenguajes de programación**

Neo4j posee un gran número de driver para utilizar, cubriendo los lenguajes más populares como .Net, Java, Javascript, Python, Ruby, PHP, etc.

- **Licencia**

Neo4j está doblemente licenciado bajo GPLv3 y AGPLv3 / commercial. AGPL hace referencia a la licencia pública general de Affero. Es una licencia copyleft derivada de la licencia pública general de GNU diseñada específicamente para asegurar la cooperación con la comunidad en caso de que el software corra en servidores de red.

Investigación

En la parte práctica, implementaremos bases de datos NoSQL para solucionar diferentes situaciones y analizaremos comparativamente sus ventajas frente a las de una base de datos relacional.

Neo4J

En primer lugar, nos propusimos investigar la posibilidad de implementar un árbol de decisión para realizar alertas basándonos en las variables ambientales provenientes de los sensores en una base de datos orientada a grafos.

Un árbol de decisión es un modelo de predicción que es utilizado a la hora de analizar la toma de decisiones. Los elementos que constituyen este árbol son, por un lado, los nodos o momentos en los que se toma una decisión por sobre las demás opciones posibles y, por otro, las aristas o uniones entre un nodo y otro que representan distintas acciones. Estos elementos se asemejan a los nodos y relaciones de un grafo ya que los mismos poseen determinadas propiedades y posibilitan las consultas sobre dichas relaciones.

Con estos lineamientos teóricos definidos, implementaremos un árbol de decisión con Neo4J para crear un sistema de alarmas en base a los datos que nos llegan desde los sensores: temperatura, luminosidad y sonido.

Estimaremos el caso ficticio de una empresa metalúrgica donde los empleados trabajan en turnos de 8 horas diarias, 6 días a la semana (un total de 48 hs. semanales) lo que podría considerarse como un trabajo continuo. La posición de trabajo es de pie y el trabajo con el cuerpo es de intensidad moderada por lo que, al aplicar la fórmula de estimación de calor metabólico, el resultado es:

$$MB + MI + MII = M$$

$$70W + 42W + 350W = 462W$$

lo que considera que es un tipo de trabajo pesado y la temperatura máxima límite es de 30ºC.

En cuanto a la iluminación, el anexo 4 de la ley N° 19.587 indica para tareas del ámbito metalúrgico, en el sector “Máquinas, herramientas y bancos de trabajo”, lo siguiente:

Tipo	Lumen
Iluminación localizada para trabajos delicados en banco o máquina, verificación de medidas, rectificación de piezas de precisión	1000
Trabajo de piezas pequeñas en banco o máquina, rectificación de piezas medianas, fabricación de herramientas, ajuste de máquinas	500

En materia acústica, la ley indica que para una exposición diaria de 8hs el nivel máximo permisible es de 135 dB(A) para sonidos continuos, utilizando protectores individuales.

Con estos datos procedemos a crear el árbol de decisión:

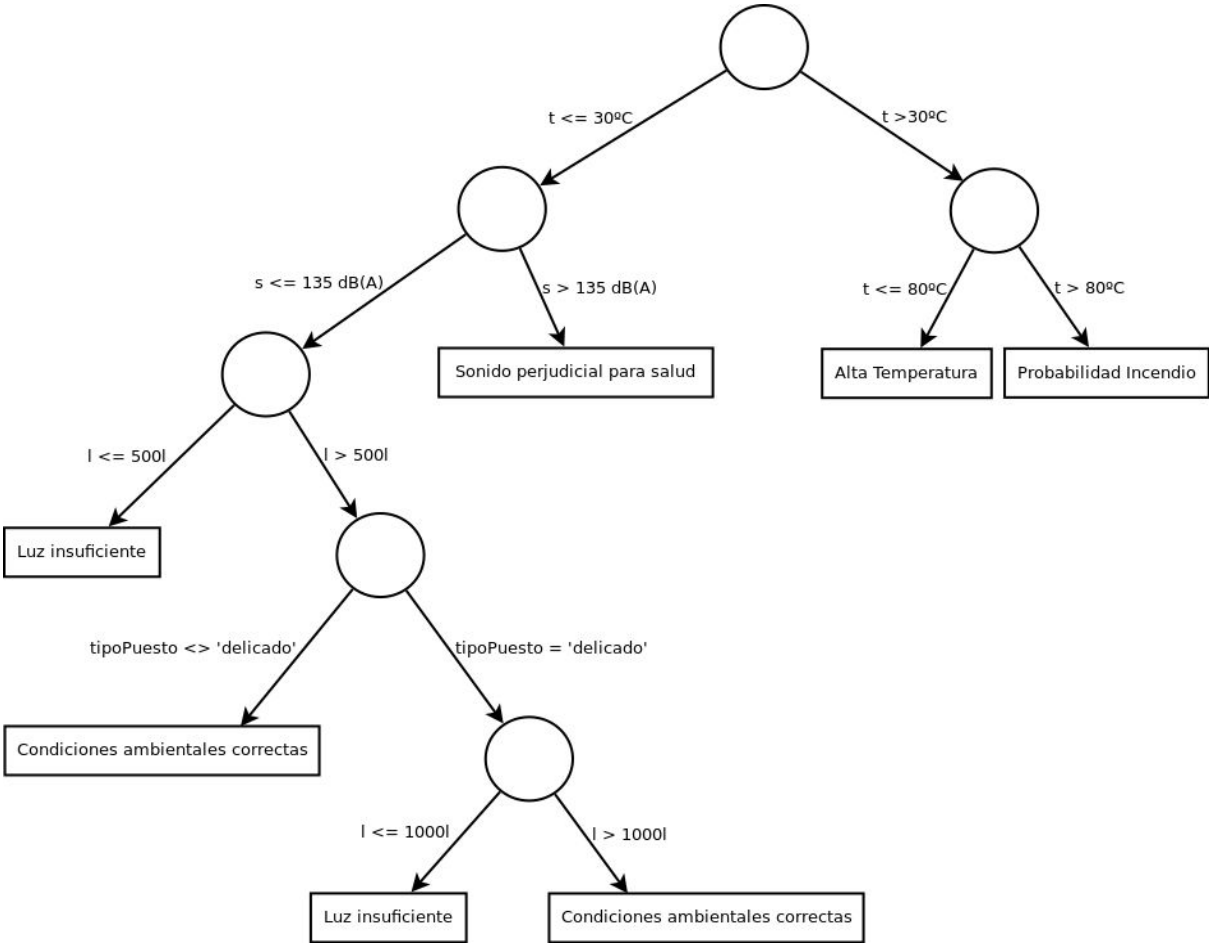

Figura 1: Grafo de ejemplo para la toma de decisión. (Fuente: los autores)

Implementación del modelo en Neo4J

Para comenzar a desarrollar la base de datos, elegimos el sistema operativo Ubuntu 16.04 LTS (long term support). El primer paso es conectarse al shell del motor de base de datos con el comando `./bin/cypher-shell -u [user] -p [password]` en la consola.

Para crear los nodos y relaciones en Neo4J, elegimos la cláusula MERGE de cypher. Esta cláusula busca entre los nodos existentes aquellos que cumplen con el criterio de búsqueda y los enlaza. En caso de que no existan, los crea para poder enlazarlos. MERGE es una combinación de las cláusulas MATCH y CREATE.

En el caso de los nodos, el patrón que se utiliza es `MERGE (node:L1:...:Ln {p1:v1, ..., pn:vn})` donde *L* son las etiquetas (labels), *p* son propiedades y *v* los valores que caracterizan a cada nodo.

En el caso de las relaciones, el patrón que se utilizamos es `MERGE (n1)-[:L1:...:Ln {p1:v1, ..., pn:vn}]->(n2)` donde n_1 es el nodo de partida, n_2 es el nodo de llegada, *L* son las etiquetas (labels), *p* son las propiedades y *v* los valores de la nueva relación.

Con el fin de identificar los distintos nodos, elegimos una jerarquía de etiquetas donde todos los nodos tiene la etiqueta *Nodo*. La etiqueta *Alternativa* la agregamos en los nodos donde se han de tomar una decisión entre dos posibles. La etiqueta *Terminal*, se encuentra en los nodos que indican un resultado definitivo. Para que cada nodo sea único, también agregamos dos propiedades llamadas *nivel* e *índice* que indica unívocamente cada nodo del árbol. En los nodos terminales también se encuentra la propiedad *resultado* que nos indica el resultado final según los datos de entrada.

En el caso de las relaciones sólo tienen la etiqueta *Decisión*. Las propiedades son tipo, valor y propiedad, que son las necesarias para evaluar los valores de entrada. Para que el árbol de decisión evalúe todos los parámetros, se agrego un tipo de relación con la propiedad `obligatorio: true`, que va desde un nodo terminal a uno alternativo y así continúa con el análisis de las variables ambientales.

Procedemos a crear los nodos y las relaciones en base al árbol de decisión modelado anteriormente y la sintaxis previamente descrita. Los comandos para crearla son los siguientes:

```
MERGE (_0:Nodo:Alternativa {nivel:0, indice:0})
MERGE (_1a:Nodo:Alternativa {nivel:1, indice:0})
MERGE (_1b:Nodo:Alternativa {nivel:1, indice:1})
MERGE (_2a:Nodo:Alternativa {nivel:2, indice:0})
MERGE (_2b:Nodo:Terminal {resultado:"Sonido perjudicial para la
salud", nivel:2, indice:1})
MERGE (_2c:Nodo:Terminal {resultado:"Alta temperatura", nivel:2,
indice:2})
```

```
MERGE (_2d:Nodo:Terminal {resultado:"Probabilidad de incendio",
nivel:2, indice:3})
MERGE (_3a:Nodo:Terminal {resultado:"Luz insuficiente", nivel:3,
indice:0})
MERGE (_3b:Nodo:Alternativa {nivel:3, indice:1})
MERGE (_4a:Nodo:Terminal {resultado:"Condiciones ambientales
correctas", nivel:4, indice:0})
MERGE (_4b:Nodo:Alternativa {nivel:4, indice:1})
MERGE (_5a:Nodo:Terminal {resultado:"Luz insuficiente", nivel:5,
indice:0})
MERGE (_5b:Nodo:Terminal {resultado:"Condiciones ambientales
correctas", nivel:5, indice:1})

MERGE (_0)-[:Decision {tipo:'menor', valor: 30,
propiedad:'grados'}]->(_1a)
MERGE (_0)-[:Decision {tipo:'mayor', valor: 30,
propiedad:'grados'}]->(_1b)
MERGE (_1a)-[:Decision {tipo:'menor', valor: 135,
propiedad:'decibelios'}]->(_2a)
MERGE (_1a)-[:Decision {tipo:'mayor', valor: 135,
propiedad:'decibelios'}]->(_2b)
MERGE (_1b)-[:Decision {tipo:'menor', valor: 80,
propiedad:'grados'}]->(_2c)
MERGE (_1b)-[:Decision {tipo:'mayor', valor: 80,
propiedad:'grados'}]->(_2d)
MERGE (_2a)-[:Decision {tipo:'menor', valor: 500,
propiedad:'lumenes'}]->(_3a)
MERGE (_2a)-[:Decision {tipo:'mayor', valor: 500,
propiedad:'lumenes'}]->(_3b)
MERGE (_2b)-[:Decision {obligatorio: true}]->(_2a)
MERGE (_2c)-[:Decision {obligatorio: true}]->(_1a)
MERGE (_2d)-[:Decision {obligatorio: true}]->(_1a)
MERGE (_3b)-[:Decision {tipo:'izquierda', valor: 'caldera',
propiedad:'tipoPuesto'}]->(_4a)
MERGE (_3b)-[:Decision {tipo:'derecha', valor: 'delicado',
propiedad:'tipoPuesto'}]->(_4b)
MERGE (_4b)-[:Decision {tipo:'menor', valor: 1000,
propiedad:'lumenes'}]->(_5a)
MERGE (_4b)-[:Decision {tipo:'mayor', valor: 1000,
propiedad:'lumenes'}]->(_5b)
```

Cuando esta serie de comandos corre exitosamente devuelve el siguiente mensaje: “Added 26 labels, created 13 nodes, set 72 properties, created 15 relationships, completed after 39 ms.”, lo que nos indica la cantidad de nodos, relaciones, propiedades y etiquetas creadas.

Ingresando a la url `http://localhost:7474/browser/` tenemos la posibilidad de ver en forma gráfica la base de datos.

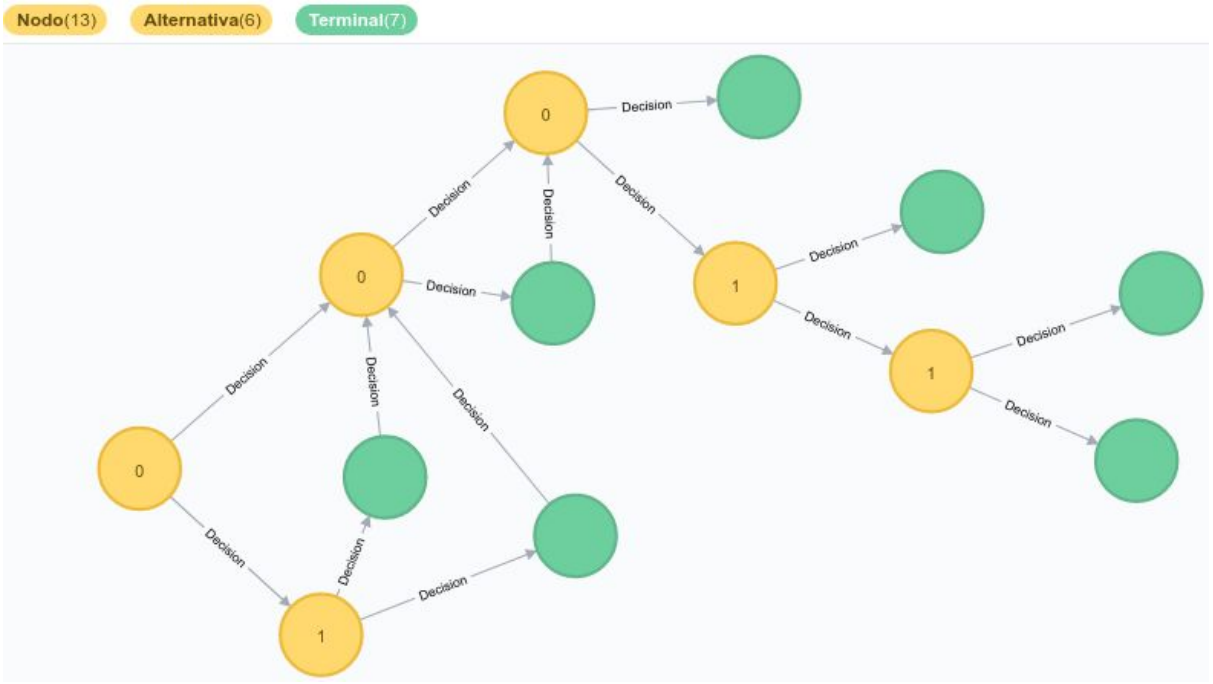

Figura 2: Grafo de la base de datos Neo4J. (Fuente: los autores)

Consultas Cypher Query Language

Con el fin de saber cuáles son los posibles caminos que los datos de entrada pueden tomar, procedemos a hacer una consulta que debe retornar todos los nodos terminales que cumplen con las siguiente condiciones:

- El nodo inicial, que es aquel que tiene las etiquetas *Nodo* y *Alternativa,* tiene las propiedades nivel 0 e indice 0.
- Está conectado mediante n relaciones con la etiqueta *Decision*.
- El Nodo final tiene las etiquetas *Nodo* y *Terminal*.

Uno de estos n nodos que resuelva la consulta será la situación actual del puesto de trabajo en el momento que se censaron las variables ambientales. La consulta que armamos es `match n = (Nodo:Alternativa {indice: 0, nivel: 0})-[r:Decision*]->(t:Nodo:Terminal) return n;` y da como resultado los 29 caminos posibles.

Para filtrar con datos de entrada, cypher provee la cláusula WITH, que luego permite utilizar los valores en el predicado de la cláusula WHERE que nos será de utilidad a la hora de evaluar el camino a tomar. La tupla de entrada tendrá un formato `{lumenes: [number], grados: [number], decibelios: [number], tipoPuesto: [string]}`.

Finalmente, para validar que todas las relaciones cumplan con las condiciones utilizaremos la cláusula WHERE junto con la función ALL, la cual evalúa que se cumpla el predicado dado para todos los elementos de la lista, que en nuestro caso la lista serán las relaciones donde se debe tomar una decisión. La consulta la escribimos de la siguiente manera:

```
ALL (r in relationships(p) WHERE
 (r.tipo = 'mayor' AND Parametros[r.propiedad] > r.valor) OR
 (r.tipo = 'menor' AND Parametros[r.propiedad] <= r.valor) OR
 (r.tipo = 'derecha' AND Parametros[r.propiedad] = 'delicado') OR
 (r.tipo = 'izquierda' AND Parametros[r.propiedad] <> 'delicado')
 OR (r.obligatorio = true)
)
```

Basta que una de las cinco condiciones se cumpla para considerar la relación válida.

Si integramos las tres consultas en una sola y retornamos la propiedad *resultado* de los nodos terminales se llega a la siguiente consulta:

```
WITH {lumenes:[],grados:[],decibelios:[],tipoPuesto: []} as Parametros
MATCH p =
(S:Nodo:Alternativa{nivel:0,indice:0})-[:Decision*]->(T:Nodo:Terminal)
WHERE ALL(r in relationships(p) WHERE
 (r.tipo = 'mayor' AND Parametros[r.propiedad] > r.valor) OR
 (r.tipo = 'menor' AND Parametros[r.propiedad] <= r.valor) OR
```

```
 (r.tipo = 'derecha' AND Parametros[r.propiedad] = 'delicado') OR
 (r.tipo = 'izquierda' AND Parametros[r.propiedad] <> 'delicado') OR
 (r.obligatorio = true)
)
RETURN T.resultado
```

Prueba en consola y análisis de resultados

Para probar la consulta, codificamos un pequeño script en NodeJS, utilizando el driver de Neo4J, que permite recibir por parámetro las variables lúmenes, grados, decibelios y tipoPuesto. El script simplemente retorna la propiedad *resultado* del nodo terminal. El script es el siguiente:

```
const neo4j = require('neo4j-driver').v1;
const argv = require('yargs').argv

const uri = 'bolt://localhost:7687'
const user = 'neo4j'
const password = 'root'
const {lumenes = 0, grados = 0, decibelios = 0, tipoPuesto =
'delicado'} = argv

const driver = neo4j.driver(uri, neo4j.auth.basic(user,
password));
const session = driver.session();

session
 .run(`WITH {lumenes: ${lumenes}, grados: ${grados}, decibelios:
${decibelios}, tipoPuesto: '${tipoPuesto}'} as Parametros
 MATCH p =
(S:Nodo:Alternativa{nivel:0,indice:0})-[:Decision*]->(T:Nodo:Termi
nal)
 WHERE ALL(r in relationships(p) WHERE
 (r.tipo = 'mayor' AND Parametros[r.propiedad] > r.valor) OR
 (r.tipo = 'menor' AND Parametros[r.propiedad] <= r.valor) OR
 (r.tipo = 'derecha' AND Parametros[r.propiedad] = 'delicado') OR
 (r.tipo = 'izquierda' AND Parametros[r.propiedad] <> 'delicado')
 OR (r.obligatorio = true)
 )
 RETURN T.resultado`)
```

```
 .subscribe({
  onNext: function (record) {
  console.log(record.toObject()['T.resultado']);
  },
  onCompleted: function () {
  session.close();
 }
})
```

Si lo ponemos a prueba en la terminal de ubuntu retorna:

```
$ node index.js --lumenes=1001 --grados=20 --decibelios=130
--tipoPuesto='caldera'
Condiciones ambientales correctas

$ node index.js --lumenes=450 --grados=20 --decibelios=130
--tipoPuesto='caldera'
Luz insuficiente

$ node index.js --lumenes=999 --grados=20 --decibelios=130
--tipoPuesto='caldera'
Condiciones ambientales correctas

$ node index.js --lumenes=999 --grados=20 --decibelios=130
--tipoPuesto='delicado'
Luz insuficiente

$ node index.js --lumenes=1001 --grados=20 --decibelios=80
--tipoPuesto='delicado'
Condiciones ambientales correctas

$ node index.js --lumenes=435 --grados=31 --decibelios=130
--tipoPuesto='delicado'
Alta temperatura
Luz insuficiente

$ node index.js --lumenes=1001 --grados=81 --decibelios=70
--tipoPuesto='delicado'
Probabilidad de incendio

$ node index.js --lumenes=500 --grados=81 --decibelios=300
--tipoPuesto='delicado'
Probabilidad de incendio
Sonido perjudicial para la salud
Luz insuficiente
```

Chequeando las salidas de consola contra el árbol de decisión, comprobamos que las relaciones funcionan a la hora de inferir la respuesta en base a las entradas.

En una base de datos relacional, cada etiqueta del nodo es una tabla, cada nodo es una tupla en dicha tabla y las propiedades de los nodos son columnas de la tabla. En el caso de las relaciones, se representan tablas intermedias cuya clave primaria son las claves de las tablas que se quieren relacionar, como se muestra en la figura siguiente:

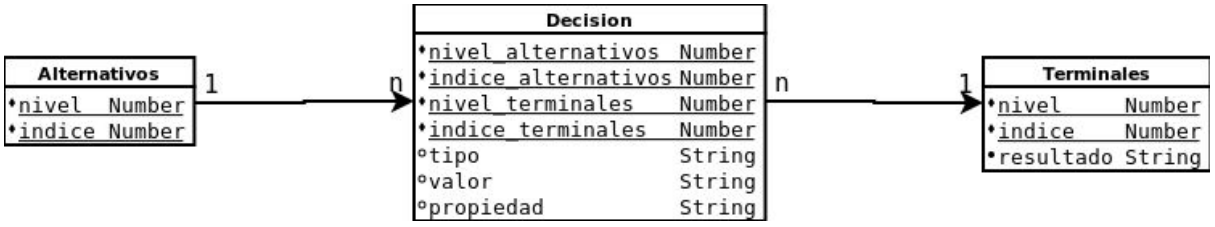

Figura 3: Modelo relacional para resolver el problema planteado anteriormente. (Fuente: los autores)

La consulta necesaria para relacionar las tablas constaría de varios JOIN entre las tablas *Alternativos*, *Decision* y *Terminales* con el fin de encontrar los posibles caminos, lo cual hace más compleja la representación y posterior búsqueda haciendo foco en las relaciones.

La facilidad de las bases de datos orientadas a grafos, y Neo4J en particular, para modelar las relaciones y luego realizar búsquedas haciendo foco en sus propiedades, hace que implementar un árbol de decisión sea más sencillo y dinámico que con una base relacional.

Redis

En segundo lugar, nos propusimos investigar la posibilidad de implementar Redis como un mecanismo de caché y validación previo a persistir definitivamente los datos sensados en otra base de datos.

Un caso de uso posible es implementar un tablero de control de los datos sensados, donde se prioriza que los datos estén disponibles rápidamente, y luego utilizar una segunda base de datos para guardarlos definitivamente. Redis también posee un lenguaje de scripting embebido llamado Lua que lo utilizaremos a la hora de validar datos anómalos y descartarlos.

Tendremos dos validaciones. La primera es chequear que el dato ingresado sea un número. La segunda validación es más compleja y se trata de que, si la posición del valor que se está ingresando lo llamamos *p*, entonces debemos chequear que el valor en *p-1* este contenido entre los valores de *p* y *p-2*.

Implementaremos el servicio pub/sub de Redis, el cual es una implementación del patrón publisher/subscriber que se utiliza típicamente para mensajería, para informarle al usuario que el sensor arrojó un valor anómalo.

Seguiremos la investigación suponiendo que en la empresa metalúrgica citada en la investigación anterior, posee un sector industrial que consta de distintas áreas, donde cada una contiene puestos de trabajo y cada puesto es un nodo de la red que envía la tupla de datos sensados.

Implementación del modelo en Redis

Para comenzar a desarrollar la base de datos, elegimos el sistema operativo Ubuntu 16.04 LTS (long term support). El primer paso es conectarse al shell del motor de base de datos con el comando `redis-cli` en la consola.

Para diseñar la key, Redis propone un esquema donde se definen distintos ámbitos separándolos con el caracter ‘:’. Por lo que en nuestro caso vamos a estructurar la clave con el siguiente patrón `[Id. area]:[Id. puesto]:[Nro. secuencial]`. El motivo de tener un número secuencial es para identificar unívocamente cada registro. Decidimos guardar el área y puesto en la key, para agilizar el proceso de búsqueda de datos y facilitar la tarea de comparar los registros que vamos a implementar posteriormente.

En el caso del value vamos a utilizar un tipo de dato llamado Hashes. Este tipo de dato permite guardar pares clave-valor dentro del value, por lo que es el modelo que más se ajusta a la tupla de datos. En nuestro caso, tendrá la siguiente estructura `lumenes [lumenes] grados [grados] decibelios [decibelios] tipoPuesto [tipoPuesto]`.

Desarrollo del script en Lua

Para comenzar a desarrollar el script primero debemos definir cómo vamos a recibir los datos sensados por parámetro, ya que el resto del desarrollo depende de su orden. Por esto definimos la siguiente estructura `redis-cli --eval [script] [area] [puesto de trabajo] [lúmenes] [grados] [decibelios] [tipo de puesto]`.

Luego vamos a crear un nuevo registro para guardar el número secuencial según el área y puesto. Primero debemos validar que el registro exista, para esto crearemos una nueva función llamada *nuevoNroSecuencial*, que tomará como parámetro el área y puesto concatenados. Con el comando GET validamos que el registro exista, si no existe lo crearemos con el comando SET, pasando como parámetro `[Id. area]:[Id. puesto]`. Luego utilizaremos el comando INCR con la key creada previamente, que nos servirá para incrementar el value del registro y lo retornaremos a una variable local del script. La función finalizada es:

```
local function nuevoNroSecuencial (areaYPuesto)
 if (redis.call('GET', areaYPuesto) == nil) then
  redis.call('SET', areaYPuesto, 0)
 end
 return redis.call('INCR', areaYPuesto)
end
```

Para agregar nuevos registros elegimos usar el comando HSET. Este comando recibe como parámetro la key, el campo, que se va a agregar si no existe y si no lo modifica, y el value. Para poder reutilizar este comando, definimos la función *agregarCampo* que simplemente toma los tres parámetros necesarios y los ejecuta el comando HSET.

```
local function agregarCampo (clave, campo, valor)
 redis.call('HSET', clave, campo, valor)
end
```

Para comenzar a realizar la validación definimos un array con los parámetros con el objeto de realizar un bloque de iteración y validar cada dato correctamente. Para consultar los registros en búsqueda del campo dentro del hash, utilizaremos el comando HGET que recibe como parámetro el index y el campo.

Por otra parte, para validar si el dato censado es un número, utilizaremos dos funciones de Lua llamada *tonumber* y *type*. *tonumber* convierte el parámetro en número si es posible hacerlo, y si no retorna *nil*. *type* retorna el tipo de dato del parámetro. Por ende, si lo anidamos se puede ver si el dato censado llegó con el formato correcto. Para informarle al usuario, utilizaremos el comando PUBLISH, al cual debemos pasarle el parámetro canal, en nuestro caso *error*, y el texto.

```
if (type(tonumber(valorActual)) ~= 'number') then
   redis.call('PUBLISH', 'error', 'Valor anomalo
'..parametros[indice]..' en '..proximaKey..'.')
```

Luego vamos a validar que el valor *p-1* ingresado este contenido en el rango entre *p-2* y *p*. Primero debemos validar que los tres valores están definidos para poder compararlos mediante `valorIndiceMenosDos ~= false and valorIndiceMenosUno ~= false and valorActual ~= false`. Luego, vamos a validar que el valor *p-1* no está en el rango mediante `not (tonumber(valorIndiceMenosUno) >= math.min(valorIndiceMenosDos, valorActual) and tonumber(valorIndiceMenosUno) <= math.max(valorIndiceMenosDos, valorActual))`, entonces si el valor no está en el rango, vamos eliminar el valor en la tupla con `redis.call('HDEL', areaYPuesto..':'..indiceMenosUno, parametros[indice])`, y luego agregar los valores anómalos en un nuevo registro mediante el comando HSET.

Si integramos todas las consultas descritas previamente, el script será el siguiente:

```
local parametros = {'area', 'puesto', 'lumenes', 'grados',
```

```
'decibelios', 'tipoPuesto'}

local function nuevoNroSecuencial (areaYPuesto)
 if (redis.call('GET', areaYPuesto) == nil) then
  redis.call('SET', areaYPuesto, 0)
 end
 return redis.call('INCR', areaYPuesto)
end

local function agregarCampo (clave, campo, valor)
 redis.call('HSET', clave, campo, valor)
end

local areaYPuesto = KEYS[1]..':'..KEYS[2]
local proximoIndice = nuevoNroSecuencial(areaYPuesto)
local proximaKey = areaYPuesto..':'..proximoIndice

for indice = 3, 6 do
 local valorIndiceMenosDos = redis.call('HGET',
areaYPuesto..':'..proximoIndice - 2, parametros[indice])
 local valorIndiceMenosUno = redis.call('HGET',
areaYPuesto..':'..proximoIndice - 1, parametros[indice])
 local valorActual = KEYS[indice]

 if (indice ~= 6) then
  if (type(tonumber(valorActual)) ~= 'number') then
   redis.call('HSET', proximaKey..':anomalo', parametros[indice],
valorActual)
   redis.call('PUBLISH', 'error', 'Valor anomalo
'..parametros[indice]..' en '..proximaKey..'.')
  else
   if (
    valorIndiceMenosDos ~= false and valorIndiceMenosUno ~= false
and valorActual ~= false and
    not (tonumber(valorIndiceMenosUno) >=
math.min(valorIndiceMenosDos, valorActual) and
     tonumber(valorIndiceMenosUno) <=
math.max(valorIndiceMenosDos, valorActual))
   ) then
    local indiceMenosUno = proximoIndice - 1
    redis.call('HDEL', areaYPuesto..':'..indiceMenosUno,
```

```
parametros[indice])
    redis.call('HSET',
areaYPuesto..':'..indiceMenosUno..':anomalo', parametros[indice],
valorIndiceMenosUno)
    redis.call('PUBLISH', 'error', 'Valor fuera de rango
'..parametros[indice]..' en
'..areaYPuesto..':'..indiceMenosUno..'.')
   end
   agregarCampo(proximaKey, parametros[indice], valorActual)
  end
 else
  agregarCampo(proximaKey, parametros[indice], valorActual)
 end
end
```

Prueba en consola y análisis de resultados

Para probar el script, primero debemos suscribirnos al canal *error* con el objetivo de recibir la notificación que el script detectó un valor anómalo. Primero debemos conectarnos al CLI con el comando `redis-cli` y luego nos suscribimos con el comando `SUBSCRIBE error`.

Vamos a tomar el siguiente conjunto de datos para poner a prueba las validaciones:

```
redis-cli --eval insert.lua 1 1 600 20 300 delicado
redis-cli --eval insert.lua 1 1 615 abc 320 delicado
redis-cli --eval insert.lua 1 1 615 25 350 delicado
redis-cli --eval insert.lua 1 1 630 24 310 delicado
redis-cli --eval insert.lua 1 1 617 23 400 delicado
```

Cuando los ejecutamos en la consola, podemos ver en el canal de errores que las validaciones funcionan:

```
"Valor anomalo grados en 1:1:2."
"Valor fuera de rango decibelios en 1:1:3."
"Valor fuera de rango lumenes en 1:1:4."
```

Si investigamos en los registros con el comando HGETALL podemos ver que el script elimina los valores anómalos en el caso del registro 1:1:3 y 1:1:4 y en el caso del registro 1:1:2 directamente no lo agrego a hash:

```
> HGETALL 1:1:2
"lumenes" "615"
"decibelios" "320"
"tipoPuesto" "delicado"
> HGETALL 1:1:3
"lumenes" "615"
"grados" "25"
"tipoPuesto" "delicado"
> HGETALL 1:1:4
"grados" "24"
"decibelios" "310"
"tipoPuesto" "delicado"
```

Además podemos ver con el comando `KEYS *:anomalo` que los siguientes registros contienen valores anómalos:

```
> Keys *:anomalo
 "1:1:4:anomalo"
 "1:1:3:anomalo"
 "1:1:2:anomalo"
```

Y luego investigando los con el comando HGETALL:

```
> HGETALL 1:1:2:anomalo
 "grados"
 "abc"
> HGETALL 1:1:3:anomalo
 "decibelios"
 "350"
> HGETALL 1:1:4:anomalo
 "lumenes"
 "630"
```

Chequeando las salidas de consola, comprobamos que tanto las validaciones como el servicio de notificaciones funcionan a la hora de leer una nueva tupla de datos sensados.

A pesar de que en la comunidad de NoSQL Redis no es una base de datos usada generalmente para persistir datos finales, tiene un amplio uso como mecanismo de caché debido a su capacidad de disponibilidad de datos. Aunque tiene mecanismos de persistencia en disco, su estructura de tabla hash no es recomendado para modelos de datos que guardan relación, como nuestro caso de los datos sensados. Otra desventaja que notamos es la falta de funciones de agregado, que podrían haber sido útiles a la hora de implementar un tablero de control con mayor información utilizando sólo las funcionalidades de Redis.

Si contrastamos nuestra solución frente a una base de datos tradicional, estas no permiten persistir los datos en memoria sino en disco, lo que sería menos eficiente. A su vez, toda la validación que desarrollamos con el lenguaje embebido de Redis debería haber sido desarrollado con otro lenguaje, ya que las bases de datos relacionales no soportan otro lenguaje que no sea SQL.

A partir de lo mencionado anteriormente llegamos a la conclusión de que Redis se adapta a las necesidades que poseemos de procesamiento de datos al poder descartar datos anómalos y persistir los datos útiles. A su vez su servicio de notificaciones es poderoso a la hora de desarrollar un tablero de control. En cambio frente a la persistencia de datos no es recomendable usar la base de datos Redis, ya que no dispone la posibilidad de establecer relaciones entre los datos sensados, lo que no la hace viable en este aspecto.

Conclusión

Durante esta investigación hemos descubierto varias aproximaciones sobre cómo utilizar bases de datos NoSQL para procesar y almacenar datos de una red de sensores. Al comenzar la investigación buscamos bases de datos que se asemejen al modelo relacional, pero haciendo foco en alta disponibilidad, modelo distribuido y escalabilidad. Esta aproximación a la investigación que tuvimos en un comienzo se vio frustrada al ver que las bases de datos más populares y utilizadas no llegaban a complacer nuestras necesidades de procesamiento y almacenamiento. Allí fue cuando decidimos plantear la investigación de otra forma y no buscar por lo convencional y plantear un pensamiento distinto.

Bajo este nuevo punto de observación planteamos la siguiente problemática: Uso de disparadores para mostrar advertencias. En este orden de ideas, utilizamos Neo4J ya que este motor de base de datos se asemeja en estructura a un árbol de decisión. Estos árboles de decisiones nos permitieron implementar una consulta para que cada tupla recibida sea procesada por Neo4J y de esta forma mostrar la situación ambiental actual. La ventaja fue que nos permitió hacer foco sobre las propiedades de las relaciones y de esta forma poder llegar a conclusiones de la situación en la que se encuentran los ambientes dentro de la industria donde estan los sensores.

Luego investigamos el uso de Redis para el desarrollo de un sistema de validación en tiempo real. La mayor ventaja de este motor de base de datos es que se ejecuta en memoria, por lo que es más rápido que otros motores de bases de datos, puede usarse como almacenamiento tipo caché y el lenguaje de scripting es más eficiente. A su vez, la ausencia de esquema permite trabajar los datos sin tener la necesidad de modelar ningún esquema. Las desventajas son que al estar almacenado en memoria es volátil y la ausencia de esquema puede generar inconsistencias a la hora de almacenarlos luego de su procesamiento. También notamos que el lenguaje de consulta es limitado a la hora de realizar operaciones en el valor o consultas de agregado.

Estas aplicaciones que le dimos a las bases de datos NoSQL al contrario de ser excluyentes, pueden ser utilizadas de forma complementaria si se estructuran en un modelo de capas de servicio, donde cada base de datos sirve para un propósito específico.

Más allá de los usos que empleamos tanto con Redis como a Neo4J, creemos que para persistir los datos de forma definitiva lo mejor es contar con una base de datos relacional que cumpla con todas las propiedades ACID, o en su defecto una base de datos al estilo Cassandra que ofrece disponibilidad y tolerancia a particiones a cambio de eventuales inconsistencias entre los nodos. También comprobamos que las bases de datos SQL ofrecen lenguajes de consultas más potentes cuando se necesitan buscar datos que tienen relaciones entre sí, lo que es importante si queremos realizar búsquedas complejas en los datos sensados.

A pesar de que las bases de datos NoSQL están ganando popularidad en los últimos años, concluimos que no van a reemplazar a las bases de datos relacionales sino a complementarlas en las áreas donde SQL no llega a satisfacer las necesidades, por ejemplo Big Data, donde se almacenan grandes cantidades de datos, o desarrollo web, donde se utilizan base de datos documentales que tiene una estructura JSON. La flexibilidad o inexistencia de esquemas al momento de diseñarlas, los bajos costes y facilidad de escalar horizontalmente, licencias open source y la comunidad activa de usuarios hacen que hoy en día los desarrolladores elijan una base de datos NoSQL por sobre las tradicionales.

Bibliografia

Ley N° 19.587 de higiene y seguridad en el trabajo, Buenos Aires, Argentina, 05 de febrero de 1979.

Introducción a No-SQL, http://www.campusmvp.es/recursos/post/Fundamentos-de-bases-de-datos-NoSQL-MongoDB.aspx , 28/03/2016

Vaish, Gaurav: Getting Started with NoSQL, PACKT Publishing, United Kingdom, 2013.

Fowler, Martin: NoSQL Distilled, Pearson Education, Inc, United States of America, 2012.

Brewer, Eric A. : *Towards Robust Distributed Systems.* Portland, Oregon, July 2000 - Keynote en el simposio sobre Principios en la Computación Distribuida. Disponible en https://people.eecs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf

Strauch, Christof: *NoSQL Databases.* Stuttgart, Alemania, 2011.

Lipcon, Todd (2009): Diseño de Patrones para Bases de Datos No Relacionales. Presentación del 11/06/2009 Disponible en http://es.slideshare.net/guestdfd1ec/design-patterns-for-distributed-nonrelational-databases

Da Silva, Maxwell Davyson: *Redis Essentials*, United Kingdom, 2015.

https://docs.mongodb.com/manual

http://cassandra.apache.org/

https://neo4j.com/docs/developer-manual/3.2/

Nelson, Jeremy: *Mastering Redis,* United Kingdom, 2016.