Comparación de PostgreSQL y MongoDB en el Contexto de una Aplicación SCADA

Tomás Juárez

1. Introducción

El presente informe da cuenta del proyecto final de la cátedra Bases de Datos 1, dictada en la carrera de Ingeniería de Sistemas, el cual consiste en establecer una comparación empírica entre una base de datos relacional y otra no relacional, siendo éstas PostgreSQL y MongoDB, respectivamente.

El proyecto consiste en diseñar la organización de la información en cada base de datos, la cual será utilizada en una aplicación tipo SCADA en donde se insertarán datos recolectados por sensores en edificios -*smart buildings*- para su monitoreo y supervisión. Para acotar la complejidad se seleccionan ciertas variables a medir en un edificio permitiendo llevar a cabo un diseño simplista manteniendo el foco en las diferencias entre las bases de datos utilizadas.

2. Marco Teórico

2.1 Sistemas SCADA

Un sistema SCADA (Supervisión, Control y Adquisición de Datos) permite la gestión y control de cualquier sistema local o remoto gracias a una interfaz gráfica que comunica al usuario con el sistema [1]. Estos sistemas disponen de una capa de software que funcionan sobre ordenadores de control de producción, el cual obtiene datos de los puntos de medición necesarios mediante piezas de hardware específicas tales como PLCs (Programmable Logic Controller) y sensores para cada variable a medir. Estos datos deben almacenarse en una base de datos para conservar el histórico del sistema. Finalmente, la comunicación con el usuario debe ser lo más sencilla e intuitiva posible, con lo cual se agregan interfaces de usuario (GUI) al sistema.

2.2 Bases de Datos Relacionales

Las bases de datos relacionales son aquellas en donde la información es representada mediante tablas relacionadas. Estas tablas se descomponen en columnas, las cuales poseen un nombre asociado para referenciarlas. Cada fila es entonces una pieza de información compuesta por distintos campos de la tabla, pudiendo estar relacionada con otras tablas. Por otro lado, cada fila de la tabla estará identificada por un único campo o por una composición de ellos; este identificador es llamado clave primaria. Si no se establece una clave primaria explícitamente, todos los campos de la tabla la representan. Las relaciones entre las tablas son llevadas a cabo mediante el concepto de claves extranjeras, las cuales consisten en contener una referencia a una clave primaria de otra tabla. Éstas se rigen según los fundamentos matemáticos establecidos por el álgebra relacional y el álgebra de bolsas y poseen un

esquema que define la estructura lógica de la información en forma de tablas y cómo estas se relacionan con otras tablas. Es importante destacar que dentro de las bases de datos relacionales, el diseño de los esquemas son usualmente normalizados. Típicamente, un buen diseño alcanzará la *tercera forma normal*. Para ello se deben concretar la primera y la segunda forma normal, así como el postulado de la tercera[1].

Otro aspecto importante es que se implementan las propiedades ACID (Atomicidad, Consistencia, Aislamiento y Durabilidad), en donde se introduce el concepto de transacción. En este contexto, cada parte de la transacción se completa, o bien no se completa ninguna (atomicidad) y luego de cada transacción la base de datos permanece en un estado consistente (consistencia), siendo éstas ejecutadas en forma aislada las unas de las otras para no afectarse entre sí mismas (aislamiento) preservando de forma permanente sus modificaciones (durabilidad).

2.3 Bases de Datos Orientadas a Documentos

Las bases de datos orientadas a documentos son uno de los tipos de bases de datos NoSQL (Not-Only SQL). NoSQL se divide en diferentes grupos, las cuales pueden ser clasificadas como orientadas a grafos, de documentos, de clave-valor o de columnas. Estas bases de datos no poseen esquema alguno o bien poseen un esquema totalmente dinámico y flexible, es decir, los campos que mantienen datos son creados bajo demanda incluso aún cuando exista información ya almacenada. NoSQL se encuentra en pleno auge debido a las aplicaciones en tiempo real y por ser utilizadas en Big Data, así como también porque propician su utilización en entornos distribuidos, los cuales están presentes en la mayoría de los productos de las grandes compañías.

La organización de la información se basa en dos factores fundamentales: en principio se busca flexibilidad para acceder a la información requerida de la forma más eficiente posible y por otro lado la posibilidad de favorecer la escalabilidad horizontal. En cierta forma, se debe saber cómo se va a acceder a la información almacenada antes de diseñar la solución en términos de organización. Esto se discutirá posteriormente en detalle.

En lo que respecta a las bases de datos orientadas a documentos, cada base de datos posee colecciones y éstas a su vez contienen documentos. Si se establece una analogía con las bases de datos relacionales, una colección se corresponde a una tabla y un documento a una fila. Cada documento posee distintos campos, típicamente representados como JSON o de forma más restrictiva, en forma de BSON.

2.4 Comparación

Como punto de partida es necesario aclarar que un enfoque no reemplaza al otro. NoSQL no surgió para reemplazar a las bases de datos relacionales en la industria del software, sino para complementar o bien para ser utilizada cuando la problemática lo amerite; no todos los escenarios son idóneos para utilizar una base de datos NoSQL, así como no todos admiten una base de datos relacional.

---

[1] Formas Normales: http://www.cs.us.es/cursos/bd-2005/tema-BD-5.pdf

Las bases de datos orientadas a documentos no poseen un esquema (o bien es variante/dinámico), lo cual se debe a que cada colección puede variar en términos de la información que contiene; sus documentos pueden ser heterogéneos, agregando o quitando campos sin verificación alguna por la base de datos ya que esta tarea corresponde al programador. En contraste, las bases de datos relacionales poseen un esquema rígido que surge de un diagrama de entidades-relaciones en donde una vez definido el esquema, este no varía a menos que se eliminen y/o agregue algún objeto en tal esquema.

Las bases de datos relacionales, tal como se mencionó previamente, poseen transacciones con propiedades ACID como una de sus partes fundamentales. Sin embargo, las bases de datos orientadas a documentos implementan las propiedades ACID de forma limitada o bien no las consideran en absoluto. Esto no es arbitrario, pero tampoco tiene que ver con el hecho de que la base de datos sea no relacional; se debe al trade-off entre la escalabilidad horizontal o implementar estas propiedades. La resolución de este conflicto se basa en que la premisa fundamental es aumentar el rendimiento y la disponibilidad respecto a las bases de datos relacionales y entre éstas se encuentra la capacidad de distribución de las particiones exclusivas de una colección (*shards*) para permitir lecturas/escrituras en paralelo, así como la introducción a la redundancia y la disponibilidad en el sistema (replica-set)[2]. En este caso, cuando uno de los *shards* se ve afectado, se pueden realizar consultas sobre otra sección de la colección (las partes no afectadas). Si no existiesen shards y una terminal que contiene una colección se cae, esta queda inaccesible en su totalidad. A su vez, cada shard se divide en una sección primaria y una o varias réplicas secundarias (redundancia). Esta parte es crucial para aumentar el rendimiento del sistema, aunque si se la utiliza mal podría decrementar el rendimiento y la capacidad al punto de inhabilitarse en su totalidad. Por otro lado, PostgreSQL posee workarounds para generar shards, pero aún así las bases de datos relacionales han demostrado serias limitaciones respecto a la escalabilidad horizontal [2].

Finalmente, un punto de discrepancia entre ambos enfoques consiste en que las bases de datos relacionales tienen como parte central las ejecuciones de consultas con agrupaciones (GROUP BY) y ensambles (JOIN). En muchas bases de datos orientadas a documentos esto no es posible dado que sólo contemplan las colecciones como estructuras independientes, las cuales solo pueden ser “ensambladas” manualmente con iteración de una selección previa a cargo del programador. Sin embargo, con el paso del tiempo las bases de datos más populares orientadas a documentos han integrado soluciones para ensamblar colecciones, entre otras operaciones inherentes a los esquemas relacionales. Cabe destacar que muchos diseños en estas bases de datos contemplan la organización de las colecciones por separado junto con referencias a identificadores de los documentos de otras colecciones, justamente por la posibilidad de integrar ensambles entre ellas. Antes de ello, solo eran posibles las estructuras anidadas o uniones manuales realizando recorridos lineales. Sin embargo, las estructuras anidadas poseían problemas para recorrer los datos cuando adquirían volúmenes grandes debido a la profundidad de los documentos y sub-documentos, así como complejidad en las

[2] Sharding en MongoDB: https://docs.mongodb.com/manual/sharding/

consultas y problemas al indexar y al distribuir la información en shards. En muchos casos, las estructuras anidadas imposibilitaba la distribución.

2.5 RDBMS y NoSQL en el Diseño de Sistemas de Software

Más allá de las restricciones de diseño impuestas por los stakeholders, ambas soluciones con todas sus variantes están presentes a la hora de cubrir atributos de calidad en un sistema de software. Si un atributo de calidad es la escalabilidad, entonces se debe tener en cuenta que las bases de datos relacionales y las bases de datos NoSQL escalan verticalmente de forma trivial, pero sólo las NoSQL escalan horizontalmente. Se entiende por escalabilidad vertical la capacidad de mejorar la performance del sistema añadiendo más recursos físicos a un nodo de procesamientos, mientras que la escalabilidad horizontal se basa en la distribución añadiendo nuevos nodos de procesamiento (sharding). Nótese que no se ha hablado respecto a la velocidad, ya que es una premisa errónea creer que una base de datos NoSQL debe ser más rápida que una relacional. Esto depende del volumen de datos y del dominio en sí, con lo cual no siempre conviene utilizarlas.

Por otro lado, las bases de datos NoSQL muchas veces son utilizadas para favorecer el desarrollo en contexto de *rapid prototyping*. Esto sucede cuando un *project owner* debe presentar rápidamente a los clientes e inversores (stakeholders) una vista funcional en poco tiempo, o bien cuando los requerimientos funcionales cambian rápidamente; los esquemas de las bases de datos relacionales son rígidos y no favorecen a la agilidad en la ingeniería de software. Generalmente se comienza utilizando NoSQL por su simplicidad y en las versiones de producción se utilizan bases de datos relacionales, una vez que el dominio esté bien definido o bien cuando los stakeholders dan el visto bueno de los prototipos funcionales. Ahora bien, más allá de la utilización de frameworks ágiles para el desarrollo del software, cuando el sistema tiene cierto nivel de criticidad en la información, es conveniente utilizar una base de datos relacional justamente por la rigidez del esquema, la capacidad de utilización de disparadores, verificaciones y las características ACID. En este punto se ve que todo depende de los casos de uso, ya que la utilización de una de ellas -o las dos en conjunto- es determinada según el dominio y/o los atributos de calidad del sistema.

3. Diseño e Implementación de la Solución

3.1 Contexto

Los edificios inteligentes son aquellas construcciones en las cuales se incluye la tecnología desde la concepción del proyecto para hacer más eficiente su uso y control. Uno de los aspectos más interesantes es el de la administración de los recursos para optimizar su utilización y minimizar los costos. En este ámbito, resulta conveniente complementar la tecnología de un edificio con un sistema SCADA para permitir la gestión de ciertos recursos. A saber, el consumo de gas y electricidad, iluminacion, humo, temperatura y humedad. Se intentarán modelar los datos previamente listados en el contexto de una empresa que presta como servicio soluciones con sistemas SCADA para edificios inteligentes, en donde los datos recolectados para cada uno de ellos se almacenan en la nube, para su posterior visualización

en cada uno de los equipos terminales correspondientes a cada constructora que contrate el servicio.

3.1 Motores Seleccionados

3.1.1 PostgreSQL

La base de datos relacional seleccionada para las pruebas fue PostgreSQL. Es una de las bases de datos más utilizadas en el mundo, estando en cuarta posición[3] y uno de los motores más completos a nivel de funcionalidades, puesto que provee utilidades como cron jobs, triggers, restricciones de integridad referencial, transacciones y utiliza el estándar SQL. Además, esta fue la base de datos utilizada a lo largo de la cursada de la materia para la cual se presenta este trabajo, lo cual hace el desarrollo más ameno para el autor.

3.1.2 MongoDB

Mongo, al igual que PostgreSQL es una de las bases de datos más populares, aunque tal como se comentó antes, son de diferente naturaleza. Esta base de datos está en quinta posición en el ranking y es la más popular entre las bases de datos orientadas a documentos.

En un principio, Mongo solo proveía consultas en estructuras anidadas utilizando los operadores posicionales[4]. Esto resultaba muy complejo, dado que solo se podía acceder a dos niveles de profundidad; si una colección tiene una estructura embebida y ésta a su vez posee otra estructura embebida, sólo es posible acceder hasta este último, haciendo que los demás niveles no puedan ser accedidos mediante una sola consulta, sino manualmente haciendo un recorrido de todos los datos traídos. Esto hace que las consultas sean ineficientes y complicadas. Para solventar algunos de estos inconvenientes se implementó la técnica *MapReduce* para realizar consultas más elaboradas con un mayor rendimiento[5]. Finalmente, se implementó el framework de agregación, el cual contempla la posibilidad de *joins* y optimiza todas las consultas de *MapReduce*[6]. De hecho, muchas funciones base de mongo e incluso algunas utilidades de *MapReduce* ya están deprecadas puesto que están siendo reemplazadas poco a poco con este nuevo framework[7]. Este trabajo fue realizado utilizando este último paradigma, el cual se explicará en detalle a continuación.

3.1.2.1 Aggregation Framework

Como se describió previamente, este paradigma de procesamiento de información establece una nueva forma de retornar información desde MongoDB, la cual superó a sus predecesores (*find* y *MapReduce*) en términos de performance. Básicamente, se detalla una serie de operaciones a completar en forma de *pipeline* en donde cada sección es llamada *stage*. Es decir, en cada stage del pipeline se modifica la información y el siguiente opera

---

[3] Estadísticas de utilización de bases de datos en ranking: https://db-engines.com/en/ranking
[4] Operador posicional ($): https://docs.mongodb.com/manual/reference/operator/projection/positional/
[5] MapReduce: https://docs.mongodb.com/manual/core/map-reduce/
[6] MapReduce vs Aggregation: https://sysdig.com/blog/mongodb-showdown-aggregate-vs-map-reduce/
[7] Aggregation: https://docs.mongodb.com/manual/aggregation/

sobre estos datos. En este contexto, hay varios detalles a tener en cuenta. En primer lugar, mongo solo utiliza los índices creados cuando el stage de filtrado (*$match*) se utiliza al principio de la función de agregación. Si esa operación está en otro lugar del pipeline, no se utilizarán los índices; siempre que sea posible es mejor filtrar la información al principio de la consulta. Por otro lado, se incluyen operaciones análogas a los *joins* de las bases de datos relacionales (*$lookup*). Sin embargo, la semántica no es la misma y trae ciertas limitaciones. No se pueden declarar múltiples campos al realizar el ensamble, es decir, si queremos ensamblar dos colecciones con claves compuestas, primero debemos ensamblarlo por un campo perteneciente a la clave primaria y posteriormente se debe filtrar con *$match* en el próximo paso del pipeline. Esto decrementa la performance ya que se realiza en dos pasos, pero sigue comportándose de forma óptima respecto a *MapReduce* y al ensamble manual. Finalmente, si se realizan ensambles entre colecciones con información anidada, esta no se puede filtrar y ensambla el documento entero, junto con sus padres y sus documentos anidados, lo cual es realmente malo en lo que respecta a la memoria. Por ejemplo, supongamos que se tienen dos colecciones: *variable* y *constructora*. La colección *variable* registra a su vez mediciones de departamentos, en donde se referencia a un departamento (dentro de la colección constructora) por cada medición, la cual se puede ver a continuación:

```
{
    _id: 1,
    nombre: "electricidad",
    medicion: [
        {
            id_departamento: 1,
            ...
        },
        {
            id_departamento: 8175,
            ...
        }
        ...
    ]
},
{
    _id:2,
    nombre: "gas"
    ...
}
...
```

La colección *constructora* posee la siguiente estructura:

```
 {
     _id: 1,
     edificio: [
         {
             _id: 52,
```

```
            departamento: [
                _id: 1,
                ...
            ]
        }
    ]
},
...
```

Si se ensamblan las mediciones de la variable que registra el consumo de electricidad, con identificador 1 (*_id:1* en colección *variable*) junto con sus departamentos en la colección *constructora* ocurriría que al hacer matching con *id_departamento* en medición e *_id* en la estructura anidada de constructora (*edificio->departamento*) se ensamblará no sólo con el departamento en cuestión, sino junto con su padre (el documento *edificio*) junto con todos sus otros departamentos. Es decir, si se quisiera ensamblar el departamento con *_id:1*, lo que ocurriría es que mongo traería también la estructura que lo contiene, es decir, el edificio con *_id:52* junto con todos los departamentos que contenga en el campo *departamento* aunque esos departamentos no nos interesen en absoluto. Esto, en efecto, nos sugiere que para establecer ensambles (*$lookup*) no sirve la organización de estructuras anidadas cuando se tiene mucha información en la colección a la derecha en el ensamble puesto que no va a tener la semántica buscada y por otro lado, la memoria se llenaría rápidamente. Estas variantes de diseño u organización de la información en MongoDB no es trivial y depende totalmente de cómo se vaya a utilizar la información, a diferencia con PostgreSQL, no importa cómo se represente la información en términos de semántica, sino en cómo favorezca a las consultas que se realizarán y sobre todo al *sharding* y la performance. El diseño elegido se discutirá en las subsiguientes secciones.

3.2 Diseño Relacional

Como la empresa presta servicios a varias constructoras es necesario asociar los edificios a cada constructora; la tabla *constructora* va a ser referenciada una o muchas veces por la tabla *edificio*. Por otro lado, las variables de medición corresponden a cada departamento de un edificio. Aquí se identifican las tablas *departamento* y *medicion_departamento*, en donde cada departamento puede tener varios registros de estas variables en diferentes tiempos. Básicamente, en *medicion_departamento* se establece qué variable se mide en un determinado departamento. Entonces, tiene sentido pensar que en un momento determinado del sistema se deberían agregar nuevas variables o dejar de medir algunas; para ello se crea la tabla *variable*, la cual registra información de cada una de ellas: el nombre, un valor booleano (*activa*) para saber si esa variable se mide o no y por último un rango de valores de normalidad de la medición (*valmin* y *valmax*). Esto da flexibilidad a la hora de modificar, agregar o eliminar variables. Finalmente, la variable a medir en cada departamento, es decir, cada registro en *medicion_departamento* debe tener un valor de medición efectiva en una fecha determinada. Esa información se almacena en la tabla

medicion, en donde se establece como clave primaria el identificador del departamento, el identificador de la clave a medir y una fecha.

El esquema discutido previamente puede visualizarse a continuación [Figura 3].

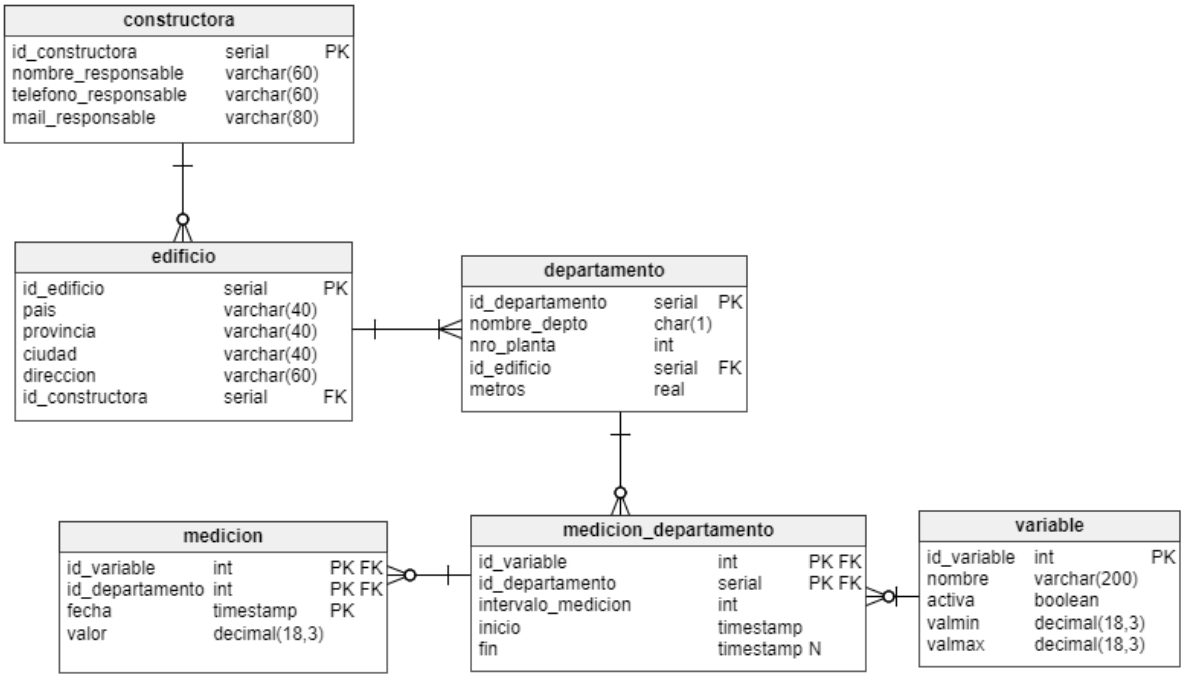

Figura 3: *Esquema relacional del sistema.*

Si bien el esquema resultante es simple, permite evaluar el rendimiento entre ambos motores de bases de datos y comprender sus diferencias fundamentales dado que es el objetivo del proyecto.

3.3 Diseño Orientado a Documentos

En el primer intento, la idea fue el anidamiento total de las entidades. Previamente se explicó que esta organización conlleva a la intervención manual del programador, lo cual lo hace ineficiente ya que el operador posicional (*$*) sólo puede utilizarse en dos niveles de profundidad desde la raíz de la colección. Además, Queda claro que el diseño de anidamiento total es un caso particular que sirve cuando se poseen pocas entidades con crecimiento esporádico de los datos en cada una de ellas o bien cuando los filtros no se realizan sobre las estructuras anidadas, sino sobre la raíz. Por ejemplo, una foto con comentarios y “me gusta”: en este caso los filtros se realizarán por foto y se devolverá junto con las estructuras anidadas sin filtro alguno. Por otro lado, la semántica del anidamiento es que los elementos embebidos no tienen razón de ser sin el documento raíz; los comentarios de la foto y los “me gusta” no deberían existir sin la foto. Este no es el caso, ya que el anidamiento total sería excesivo por la profundidad y el volumen de datos.

La segunda opción fue una organización de anidamiento parcial, en donde surgen dos escenarios posibles. En ambas opciones se crean tres colecciones: *constructora*, *variable* y *medicion*. En el primer caso [Figura 3], en la colección constructora, cada documento contiene edificios anidados, y estos a su vez poseen departamentos anidados, los cuales contienen determinadas mediciones. Estas mediciones referencian a una variable en particular a la colección *variable*, la cual contiene los identificadores de cada variable y sus datos. Finalmente, en la colección *medicion* se almacena el valor medido de una variable, un departamento y una fecha en particular; esta referencia al identificador de un departamento y al identificador de una variable se hace a la colección *constructora* (en el último nivel de anidamiento). Retomando las limitaciones de Mongo, vemos que hay 3 niveles de anidamiento en la colección *constructora*, lo cual no permite ingresar a las mediciones de un departamento mediante el operador posicional. Otro inconveniente es que para ensamblar las mediciones registradas por cada departamento con la medición efectiva del mismo en una fecha determinada (colección *medicion*) se debe buscar primero en el tercer nivel de anidamiento de la colección *constructora* lo cual agrega un overhead a la búsqueda. Las inserciones en constructora rara vez ocurren; no se insertan muchas constructoras y los edificios se agregan esporádicamente. Los departamentos se agregan todos juntos una vez que se agregue un edificio, así como las variables que se deseen medir para cada departamento. El lado positivo de este esquema es que no haría falta ensamblar con la colección *variable* cuando se desee consultar una medición de un departamento por fecha, o consultas similares, lo cual lo haría más rápido. La segunda opción es almacenar edificios y sus respectivos departamentos en la colección *constructora* y salvar las mediciones como documentos embebidos dentro de la colección *variable* para reducir el número de anidamiento en la primer colección [Figura 4]. Al separar los registros de las variables de medición en un departamento de la colección *constructora* habrá que ensamblar los departamentos con sus registros de *variable* en donde se registren sus variables a medir, para luego ensamblar con la medición efectiva en la colección *medicion*. Si bien esto presupone un ensamble más que el primer caso, al haber pocos datos en la colección variable, esto no es crítico. Sea cual sea el escenario elegido, hay que ensamblar las colecciones, con lo cual se debe utilizar *MapReduce*, el framework de agregación o bien hacerlo manualmente. Se discutieron estos tres casos previamente y se mostró que el caso del framework de agregación es el más óptimo de los tres en términos de performance. Sin embargo, para ensamblar se debe utilizar el stage *$lookup* en alguna parte del pipeline de *aggregate*, lo cual no es posible dado que en las estructuras existe mucha información anidada; como se comentó, si quisiéramos ensamblar la colección medición con 8.000.000 de registros, la colección variable con la variable necesaria (supongamos, electricidad) y posteriormente con el departamento en cuestión, no sólo nos ensamblará con el departamento que deseemos en cada caso, sino que traerá en cada ensamble el departamento y el edificio al que pertenece, y este a su vez tendrá incluido a todos los departamentos en el. Este esquema consume demasiada memoria, ya que esta se llenaría rápidamente o bien la consulta tardaría horas o días.

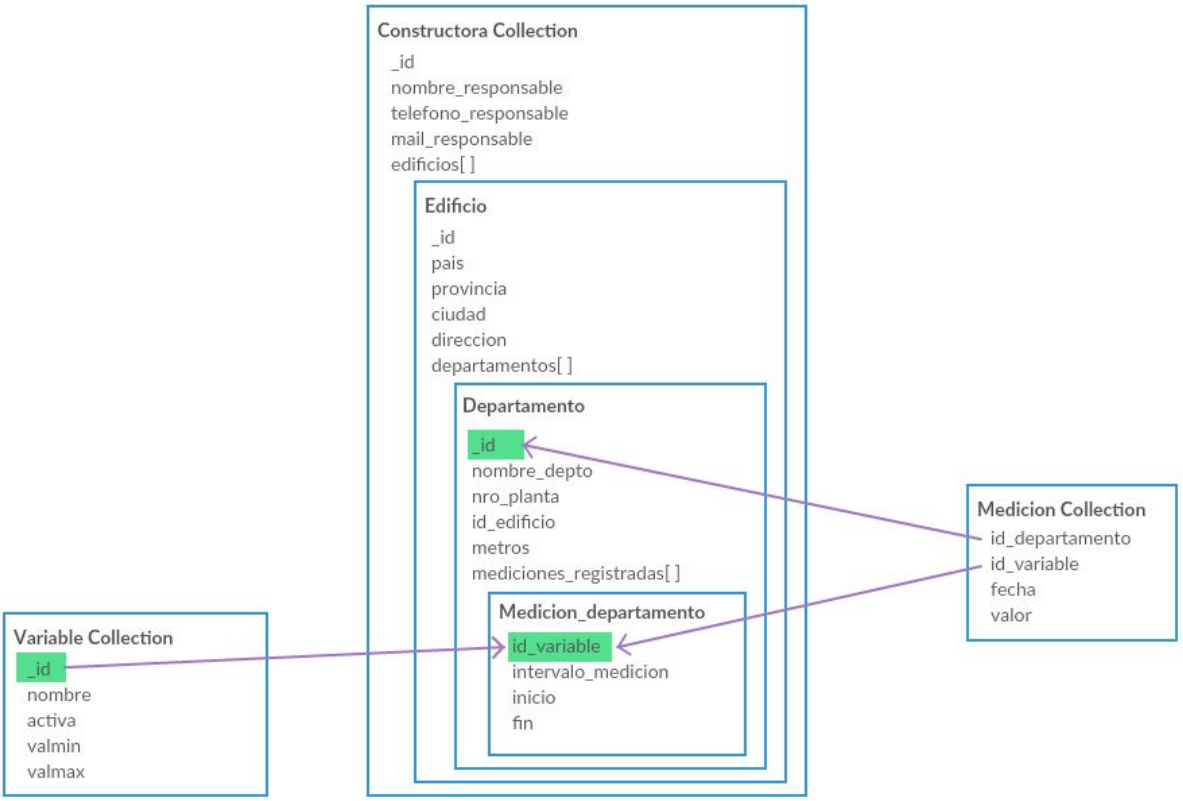

Figura 3: *diseño con tres niveles de anidamiento en constructora*.

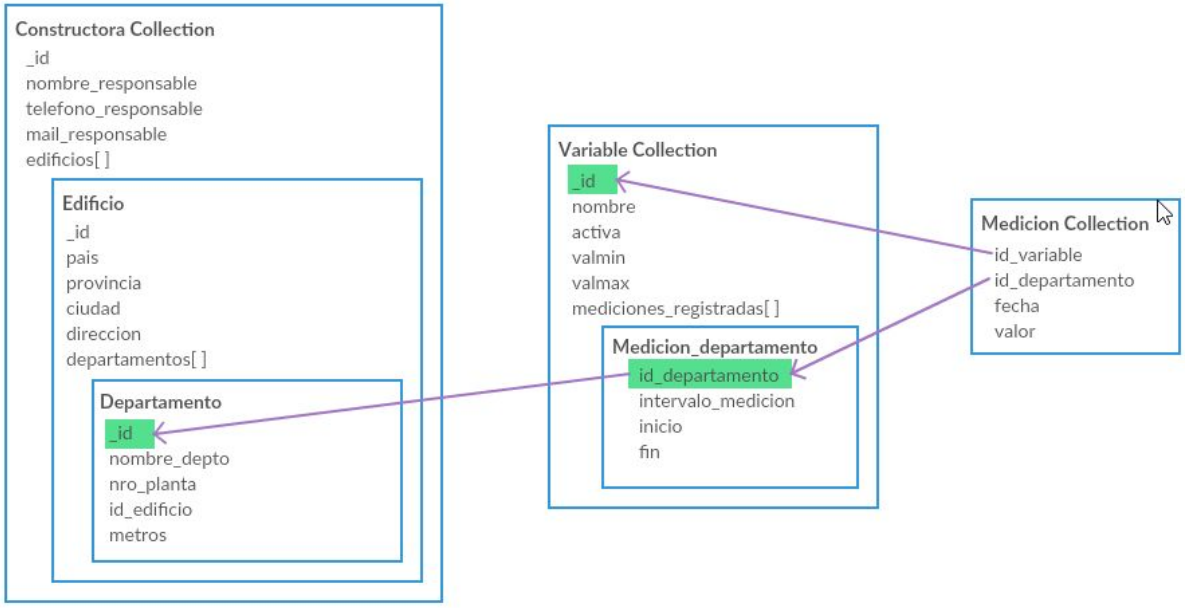

Figura 3: *diseño con máximo de dos niveles de anidamiento en constructora*.

Quedando descartados los tres esquemas anteriores, lo mejor para este caso particular es el de crear una colección para cada entidad reconocida; *constructora, edificio, departamento, medicion_departamento, variable* y *medicion*. Este es un diseño similar al del esquema relacional, ya que se incluyen referencias en las colecciones (*FK*). Sin embargo, esto

no sólo nos permite optimizar el tiempo de respuesta de las consultas, sino que permite mayor flexibilidad en las consultas, ensamblar rápidamente los documentos deseados y además favorece a la distribución o *sharding*, ya que las estructuras anidadas hacen esta tarea complicada y eventualmente ineficiente. Este diseño es el elegido para realizar las pruebas y se puede ver a continuación [Figura 4].

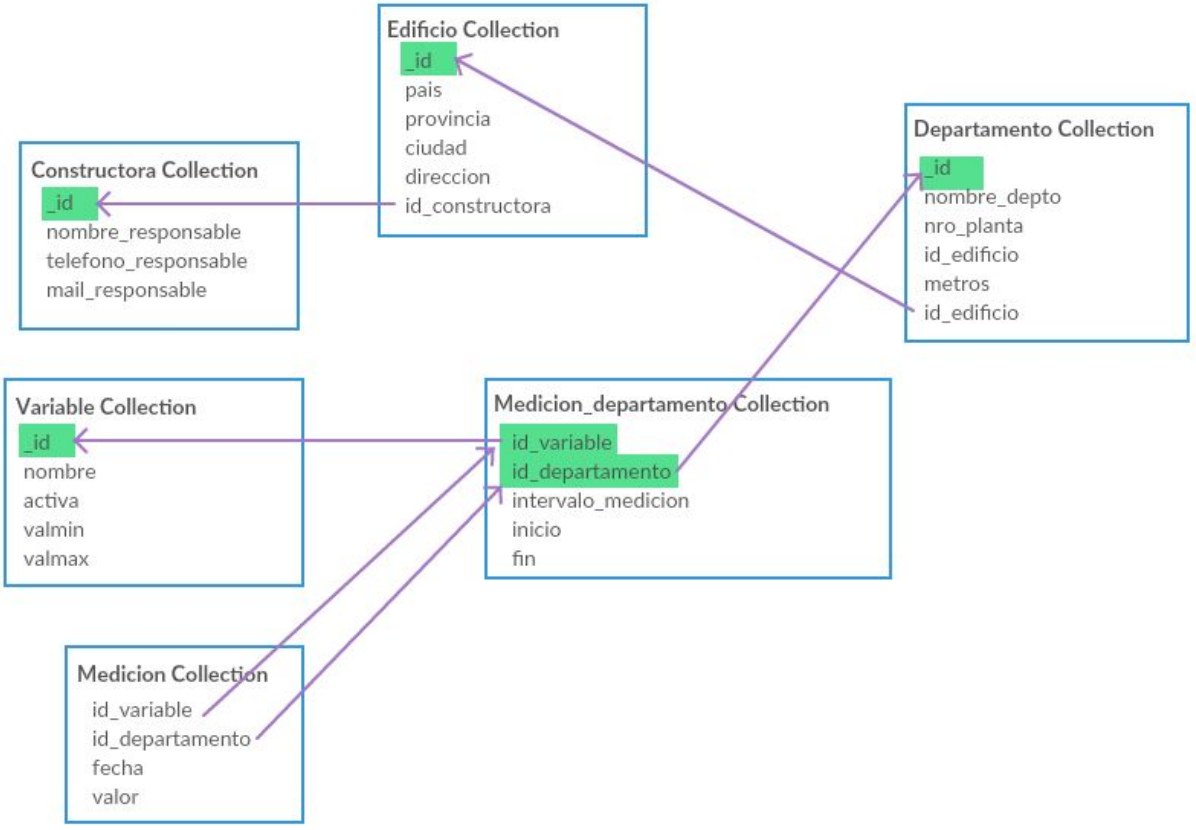

Figura 4: *diseño final*.

Cabe destacar que si bien este diseño final es equivalente al diseño relacional, la principal diferencia radica en el motor de la base de datos y en cómo procesa la información internamente, es decir, por más que el diseño sea similar al esquema relacional no quiere decir que la solución en este contexto sea relacional ya que la base de datos no es de tal naturaleza. Además, estas referencias entre claves de una colección a otra no son verificadas como restricciones, pero nos permiten tener una gran flexibilidad a la hora de concretar ensambles. Si bien el alumno comprende que generalmente es conveniente utilizar las información lo más denormalizada posible en MongoDB, este dominio no propició este tipo de diseño por las razones explicadas previamente. Este diseño fue elegido luego de un largo análisis sobre las posibilidades que permitía resolver la problemática dada, en donde la normalización fue la clave para permitir una utilización más eficiente del framework de agregación y permitir mayor facilidad respecto a la escalabilidad horizontal, pensando que a futuro podrían agregarse *shards*. En este caso, los *shards* serían convenientes sobre la colección más grande (*medicion*); ocho *shards* en donde cada uno corresponde a cada

variable o bien *sharding* en rango por años. El primer caso tiene una limitación y es que no se puede concretar *sharding* con más nodos primarios que la cantidad de variables; ocho será el número máximo. Ahora bien, en el caso de sharding con fechas, esto ayudará a expandir la distribución cada vez que se agreguen nodos primarios, pudiendo dividir en rangos más restrictivos y teniendo así una granularidad más fina para concretar esta tarea lo cual nos permite ampliar la escalabilidad horizontal incluso hasta dividiendo en el contexto de meses cuando el *sharding* sea ya indivisible por años.

Por otro lado, la colección medición se creó utilizando el campo *_id* creado por defecto como clave primaria por simplicidad. Para asegurarse que ningún departamento en la colección medición tenga una medición en una fecha determinada bajo la misma variable (repetición), se procede a crear un índice único sobre la colección con la composición *<id_variable, id_departamento, fecha>*. Esto, además, favorecerá a las consultas que utilicen estos campos. La creación de los índices se resuelve con el siguiente comando:

```
db.medicion.createIndex(
    {id_variable:1, id_departamento:1, fecha:1},
    {unique:true, name:"ix_pk_medicion"}
);
```

La primer parte declara los campos que van a formar parte del índice y el valor (1 en todos los casos) define el orden, el cual será ascendente. Por otro lado, en el segundo componente de la función *createIndex*[8] se establecen opciones, tales como la no repetición (*unique*) y el nombre identificador del índice.

4. Generación de Datos y Configuración del Entorno

La información generada fue aleatoria dentro de parámetros acordes a la situación. En primer lugar, la información fue creada en PostgreSQL para su posterior migración a MongoDB. Para realizar esta tarea, se utilizó una herramienta llamada Data Generator[9], la cual nos permite generar filas en tablas sin límite si se crea un usuario y se instala localmente. Para ello, se instaló la configuración básica LAMP para correr scripts PHP utilizando Apache Server en el contexto de linux, específicamente Ubuntu 16. Una vez en la herramienta, se procedió a crear las tablas y establecer parámetros para la creación de datos; las fechas de cada medición, las posibles variables, los rangos mínimos y máximos, entre otros. Esto conlleva mucho tiempo debido al cuidado que requiere establecer las referencias válidas entre ciertas tablas. Por eso la información fue creada por tandas de, como máximo, 50.000 filas por generación, ya que debían coincidir con los identificadores de los departamentos y esto se realizó 20 veces por cada variable en distintas fechas. Todos estos datos generados se almacenan dentro de la carpeta *postgresql/data*. Para insertar estos datos se creó un script bash en la raíz de esta misma carpeta llamado *load_data.sh*, el cual arrojará en un fichero llamado *time.txt* el tiempo tomado por las inserciones. Una vez generados en PostgreSQL, hay que migrar la información a MongoDB. No existen aún herramientas que permitan pasar

[8] CreateIndex: http://www.w3resource.com/mongodb/shell-methods/collection/db-collection-createIndex.php
[9] Data Generator: http://www.generatedata.com/?lang=es

de un esquema relacional a un diseño particular de Mongo o bien son pagas. Por este motivo, se implementó un programa en Java que permitiera conectarse a PostgreSQL e insertar la información a MongoDB, fila por fila según el diseño requerido en Mongo. Este programa se construyó en base a Maven, el cual es un sistema para gestionar dependencias entre paquetes, ya que los drivers de ambos motores son requeridos para ejecutar. Un detalle no menor es que en PostgreSQL se insertaron los datos de tipo *timestamp* sin huso horario. En el caso de MongoDB no es posible obviar el huso horario con lo cual hay que realizar la conversión al huso local (desde donde se encuentran los servidores), esto es ajustar tres horas debido a que las pruebas se realizaron desde la ciudad de Tandil (GMT -3). Una vez concretada esta tarea, se exportan las colecciones de mongo en archivos json en la carpeta *mongodb/data* utilizando el script *export.sh*; una especie de dump en MongoDB. Luego se eliminan los datos en Mongo (la base de datos completa) y se inserta la información mediante un script (*load_data.sh*) en la carpeta *mongodb/data/*, la cual toma el tiempo que conlleva la inserción y la almacena en un archivo de texto llamado *time.txt*.

Tanto en la carpeta postgreSQL como en la carpeta MongoDB existen tres carpetas dentro de ellas llamada *selects*, *updates* y *deletes*. Dentro de ellas se encuentran las consultas de selección, actualización y eliminación en los formatos necesarios para Mongo y PostgreSQL (sql y js). Las consultas de actualización y eliminación se ejecutan una sola vez dado que modifican el estado de la base de datos y para volver a ejecutarlos se debería restaurar la base de datos a sus estados anteriores, lo cual demoraría horas. Para tomar las métricas de las distintas consultas de selección, se crearon distintas carpetas dentro de /*select*; una para cada grupo de consultas (con agrupamientos y funciones de agregación, con distinct y con ensambles). Dentro de cada una de estas carpetas existe un script llamado *run.sh*, el cual ejecuta cada consulta múltiples veces (por defecto 30) y hace un promedio de las ejecuciones de cada consulta. Toda esta información es arrojada en un archivo de texto llamado *time.txt*.

Finalmente, existen también archivos de creación y eliminación de índices para Mongo y PostgreSQL bajo la carpeta llamada */indexes* y para el caso particular de PostgreSQL, existe una carpeta llamada */schema* en donde se proveen archivos de creación y eliminación del esquema definido, así como el script *prepare.sh* que crea la base de datos, el esquema, las tablas, restricciones y los índices. Esto se hace manualmente en MongoDB ya que aquí no se crea esquema alguno, sino que las colecciones toman forma cuando se les inserta información.

5. Consultas de Selección

Las consultas a ejecutar corresponden a consultas de selección y actualización. En cada una de ellas se discutirán estrategias de optimización y su análogo en ambos motores.

5.1 Consultas de Selección

Las consultas de selección pueden ser divididas, a grandes rasgos, dentro de tres grupos: consultas sin repeticiones, con funciones de agregación y agrupamiento y con

ensambles entre diferentes tablas. A continuación se darán ciertos requerimientos a resolver mediante consultas de cada uno de estos tres grupos.

5.1.1 Selección con Agrupadores y Funciones de Agregación

5.1.1.1 Consulta 1

Se desea obtener el total de consumo registrado de cada una de las variables en los registros de mediciones para fines estadísticos.

Como primera medida, al analizar este requerimiento surge la necesidad de realizar un agrupamiento por cada variable junto con su respectivo consumo. En este caso, no hay condiciones de filtrado, así que se tiene un escaneo total de la tabla de mediciones efectivas.

PostgreSQL

```
SELECT id_variable, SUM(valor) as "consumo"
FROM medicion
GROUP BY id_variable;
```

En este caso, existe una optimización posible y es la de crear un índice sobre la tabla medición en la columna *id_variable*. Sin embargo, el campo *id_variable* ya está incluido en un índice único creado por defecto por PostgreSQL por formar parte de la clave primaria *<id_departamento, id_variable, fecha>*, lo cual ayuda a que el motor no realice un escaneo total de la tabla, sino un recorrido sobre un índice *BTREE*. Existe una discusión al respecto, sobre la conveniencia de crear un índice específicamente para un campo cuando este ya existe en otro índice compuesto en donde este campo forma parte del mismo; la mejor decisión en términos de performance es crear un índice sólo para *id_variable*, pero aquí hay un trade-off entre la memoria que ocuparía y los beneficios que generaría respecto a la velocidad de las consultas[10]. Al correr esta sentencia con *EXPLAIN ANALYZE* se ve que el motor realiza un escaneo total sobre la tabla e ignora el índice que contiene el campo *id_variable*. Esto se debe a que el planificador decide si utilizar un índice o no y en el caso particular de PostgreSQL, no se puede forzar la utilización de un índice en una consulta. Sin embargo, puede que posteriormente sea utilizado para la misma consulta a medida que el planificador realice las estadísticas necesarias.

MongoDB

```
db.medicion.aggregate([
    {$group:
        {
            _id: "$id_variable",
            consumo: { $sum : "$valor" }
```

[10] Creación de índice para un campo cuando este forma parte de un índice multiclave: https://stackoverflow.com/questions/11352056/postgresql-composite-primary-key

```
        }
    }
]);
```

En este caso, no fue posible utilizar la función *db.collection.group()*, dado que en las últimas versiones de MongoDB fue deprecada dando lugar a su versión más eficiente, la cual pertenece al framework de agregación. En esta consulta las operaciones se realizan sobre la colección *medicion* y hay un único stage (*$group*), en donde se explicita la clave de agrupamiento *_id* como el campo *$id_variable*. El signo pesos que antecede al campo a seleccionar es un operador posicional. Finalmente, la función de agregación de suma se realiza mediante *$sum*. En MongoDB no es posible utilizar índices para el stage *$group* ya que los índices son útiles únicamente en los stages *$sort* y *$match* cuando se encuentran al principio del pipeline[11].

5.1.1.2 Consulta 2

Se desea obtener la cantidad de edificios en donde sus departamentos registrados suman más de 80000 metros cuadrados.

En este caso, existe una condición de filtrado y una función de agregación en la misma, ya que se debe sumar los metros de los departamentos agrupando la información por cada edificio.

PostgreSQL

```
SELECT COUNT(*) FROM (
       SELECT 1 FROM departamento
       GROUP BY id_edificio
       HAVING SUM(metros) > 80000
) edificios_filtrados;
```

En esta consulta no es necesario crear un índice sobre la clave de agrupamiento debido a que la tabla es chica en cantidad de filas (~50.000) y no es conveniente crear uno ya que rara vez crece. Por otro lado, fue necesaria una subconsulta para filtrar los edificios que tengan superen los 80000 metros cuadrados sumando sus departamentos para luego contar las filas devueltas; ese será el número de edificios que superen el valor umbral.

MongoDB

```
db.departamento.aggregate([
    {$group:
        {
            _id: "$id_edificio",
            suma: { $sum : "$metros" }
        }
```

[11] Stages que utilizan índices: https://docs.mongodb.com/master/core/aggregation-pipeline/#pipeline-operators-and-indexes

```
    },
    {$match: { "suma": { $gt : 80000 } } },
    {$count: "cantidad"}
]);
```

Aquí se opera sobre la colección departamento y se utilizan tres stages: *$group*, *$match* y *$count*. En primer lugar se agrupan a los departamentos por edificio y se utiliza un acumulador con la suma de los departamentos por edificio. Posteriormente, se filtran los documentos agrupados que no posean en "*suma*" un valor superior a 80.000. Es importante destacar que *$group* proyecta sólo los campos definidos al siguiente stage; _id (con sus composiciones, si las hay) y los acumuladores definidos. Finalmente se cuentan los documentos filtrados. El stage *$match* actúa como *having* de SQL cuando se lo utiliza luego de *$group*.

En el caso de los índices, ocurre lo mismo que en *MongoDB*: al tratarse de una colección pequeña, no hace falta agregar índices.

5.1.1.3 Consulta 3

Se desea obtener el promedio de cada variable y los máximos y mínimos registrados para cada una de ellas y ordenarla en orden decreciente según su consumo medio.

Para resolver esta consulta se deben tener en cuenta tres funciones de agregación y el agrupamiento por variables.

PostgreSQL

Las funciones de agregación de mínimos y máximos son optimizables cuando se tiene un índice sobre el campo en el cual se está operando. Sin embargo, en este caso hay una tercera función de agregación (AVG) la cual opera sobre la tabla en su totalidad por lo que no es conveniente crear un índice en el campo valor de la tabla *medicion*. Sin embargo, si la tabla tuviese muchas columnas (~20+) entonces sería provechoso crear un índice sobre este campo de forma tal que el motor realice un escaneo total sobre el índice en lugar de realizar el escaneo total de la tabla ya que esta es mucho más grande y consumiría más tiempo por un acceso ineficiente. La consulta SQL se muestra a continuación:

```
SELECT id_variable, AVG(valor) as "promedio", MIN(valor) as "mínimo", MAX(valor)
as "máximo"
FROM medicion
GROUP BY id_variable;
```

MongoDB

En el caso de MongoDB ocurre lo mismo que en PostgreSQL respecto a los índices. En esta consulta se utilizó el framework de agregación, la cual se puede ver a continuación:

```
db.medicion.aggregate([
    {$group:
        {
```

```
            _id: "$id_variable",
            minimo: { $min: "$valor" },
            maximo: { $max: "$valor" },
            promedio: { $avg: "$valor" }
        }
    }
]);
```

Aquí hay solo un stage en el pipeline, *$group*. Dentro de él se encuentran tres agrupadores o acumuladores: *$min*, *$max* y *$avg* respecto al campo *valor* siendo estos para el mínimo, el máximo y el promedio.

5.1.2 Selecciones sin Repeticiones

5.1.1.1 Consulta 4

Se desea obtener los identificadores ordenados de forma ascendente de todos los departamentos que registran mediciones en el año 2010, teniendo en cuenta que pueden existir más de una vez en los registros de mediciones.

PostgreSQL

Un primer intento para obtener esta información es utilizando la palabra clave *DISTINCT* respecto a la clave primaria de *departamento*, la cual se repite múltiples veces en la tabla *medicion*, ya que es parte de la clave primaria de la misma.

```
SELECT DISTINCT id_departamento
FROM medicion
WHERE EXTRACT(year FROM fecha) = 2010
ORDER BY id_departamento;
```

Como puede verse en la consulta, hay una condición en donde se toma sólo los registros cuyo año sea el 2010. Como la tabla es muy grande en términos de filas, es necesario ejecutar el analizador del motor utilizando *EXPLAIN ANALYZE*. Esto conlleva un escaneo total de la tabla, con sus millones de registros con lo cual hay que optimizar la consulta y en este caso basta con crear un índice. El índice sería la parte extraída de la fecha (el año) que pertenece a la condición pero si se creara un índice por años no serviría en el caso condicionar con meses o días, por lo que conviene crear un sólo índice por fecha completa para poder utilizarlo en cada consulta que contenga una fecha.

```
CREATE INDEX ix_fecha_medicion ON medicion(fecha);
```

Surge entonces la necesidad de reescribir la consulta para poder aprovechar el índice sobre la tabla. La forma final de la selección puede verse a continuación:

```
SELECT DISTINCT id_departamento
FROM medicion
WHERE (fecha >= '2010-01-01'::date AND fecha < '2011-01-01'::date)
ORDER BY id_departamento;
```

Es importante destacar que la consulta pasó de ser una condición puntual a una condición en un rango de 12 meses, obteniendo un año puntual pero especificando la directiva desde otra perspectiva.

MongoDB

En este tipo de consultas, MongoDB provee una función para realizar consultas sin repeticiones llamada *distinct()*. De hecho, es la única función del motor para llevar a cabo esta tarea. Como primer parámetro se establece el campo por el cual se desean eliminar las repeticiones (*id_departamento*) y como segundo parámetro se explicita la condición de filtrado. En general, las funciones de mongo siguen el patrón al estilo *pipes & filters* en donde la salida de una funcionalidad retorna un cursor con la información modificada, lista para ser leída o procesada por otra función. En este caso, primero se eliminan las repeticiones según el identificador de los departamentos y luego se ordena ascendentemente según este mismo campo utilizando la función *sort()*.

```
db.medicion.distinct(
    "id_departamento",
    {"fecha":
        {
            $gte: ISODate("2010-01-01T03:00:00.0Z"),
            $lt: ISODate("2011-01-01T03:00:00.0Z")
        }
    }
).sort();
```

Para optimizar la consulta se creó un índice sobre la fecha de forma tal que la consulta en rango sobre este campo se ejecute más rápido, según el siguiente comando:

```
db.medicion.createIndex({fecha:1},{name:"ix_fecha_medicion"});
```

Los operadores *$gte* y *$lt* son comparadores por mayor e igual y menor, respectivamente. Por otro lado, existen diversas optimizaciones en el framework de agregación que harían esta consulta más eficiente. Desgraciadamente, no existe un stage *$distinct* que permita eliminar las repeticiones, pero se puede llevar a cabo un workaround ya que es posible agrupar por múltiples claves. En este caso, el agrupamiento por el identificador del departamento sin ningún otro acumulador (o función de agregación) hace que se unifiquen las repeticiones, eliminándolas, valga la redundancia. Esto presenta una mejora notable en términos de performance. La consulta optimizada utilizando el framework de agregación puede verse a continuación:

```
db.medicion.aggregate([
    {$match:
        {"fecha":
            {
```

```
                $gte: ISODate("2010-01-01T03:00:00.0Z"),
                $lt: ISODate("2011-01-01T03:00:00.0Z")
            }
        }
    },
    {$group:{_id:"$id_departamento"}},
    {$sort:{"_id":1}}
]);
```

El primer stage es el filtro de las mediciones en el año 2010, utilizando *$gte* y *$lt* para aprovechar los índices sobre la fecha. Aquí hay un detalle a tener en cuenta y es que MongoDB obliga a utilizar un huso horario en los campos *timestamp*, con lo cual debemos establecer el local (GMT -3) simplemente añadiendo *T03:00:00.0Z* en la fecha a consultar. En el segundo stage se agrupa según el campo *id_departamento* y finalmente se ordena crecientemente según el campo proyectado en el stage previo; *id_departamento*. Si se hiciera un análisis sobre esta cuestión en lo que respecta a la semántica se podría concluir que no es lo mismo agrupar que eliminar los repetidos, pero en este caso la otra alternativa es operar manualmente para eliminar restos, lo cual lo sería ineficiente.

5.1.1.2 Consulta 5

Se desea seleccionar sin repeticiones los años y meses en donde se registran mediciones de cualquier variable y cualquier departamento.

PostgreSQL

En este escenario, no se utilizan índices ya que no se tiene ningún filtro. Por otro lado, se emplea la cláusula DISTINCT con años y meses, extraídos como parte de la fecha obtenida en cada fila para no repetir información.

```
SELECT DISTINCT EXTRACT(year FROM fecha), EXTRACT(month FROM fecha)
FROM medicion;
```

MongoDB

En el caso de Mongo, no es posible utilizar la función *distinct()* ya que esta no puede eliminar repetidos según dos campos; sólo se la puede utilizar cuando se desea operar con distinct sobre un sólo campo, no sobre objetos o composiciones. Para concretar esta operación se utilizó el framework de agregación con un agrupamiento entre dos campos: mes y año como parte del identificador del stage *$group*. Esto elimina las repeticiones ya que sólo se seleccionan los dos campos agrupamiento sin otro acumulador o función de agregación. El código que resuelve la problemática se ve a continuación:
El año y mes de la fecha se extraen con los operadores *$year* y *$month* ajustándolos al huso horario local (GMT -3). Para ello se resta la extracción del mes/año respecto al offset en 10800000 milisegundos (3 horas) teniendo mediante *$substract*.

```
db.medicion.aggregate([
```

```
    {$group:
        {_id:
            {
                "año":{
                    $year:[{$subtract:["$fecha",10800000]}]
                },
                "mes":{
                    $month:[{$subtract:["$fecha",10800000]}]
                }
            }
        }
    }
]);
```

5.1.3 Selección con Ensambles

5.1.3.1 Consulta 6

Seleccionar los departamentos que registran al menos una medición de datos superiores a 90GB en el mes de mayo de 2010.

PostgreSQL

Esta consulta a la gran mayoría de las tablas del esquema relacional, ya que se opera con ensambles entre cada una de ellas. Esto es así porque no sólo hay que obtener información sobre los registros de medición de un año en particular, sino también del departamento que supere un valor umbral. El identificador de la variable es obtenido de la tabla *variable* ya que es una clave primaria de ésta tabla, y el punto de encuentro entre la tabla *departamento* y *variable* es *medicion_departamento* ya que contiene la información de los departamentos y las métricas que poseen habilitadas, es decir, las variables asociadas.

```
SELECT D.*
FROM departamento D
INNER JOIN variable V ON(v.nombre = 'datos')
INNER JOIN medicion_departamento M ON (D.id_departamento = M.id_departamento AND
V.id_variable = M.id_variable)
INNER JOIN medicion X ON (X.id_departamento = M.id_departamento AND X.id_variable
= M.id_variable)
WHERE (
       (X.valor > 90) AND
       (EXTRACT(year FROM fecha) = 2010) AND
       (EXTRACT(month FROM fecha) = 5)
);
```

Las optimizaciones en la consulta se basan en principio a ensamblar desde la tabla más chica involucrada a las más grande respecto a la cantidad de registros que estas posean, siempre que sea posible. En esta caso, la tabla *departamento* está antes de *variable* ya que si fuese al revés no habría forma de ensamblar la tabla departamento *inmediatamente*; habría

que ensamblar *medicion_departamento* antes, la cual posee 400.000 filas. Todos los ensambles se hacen por claves primarias y esto presupone la utilización de los índices que PostgreSQL crea por defecto por cada uno de ellos, optimizando entonces la performance. El único caso en el que esto no ocurre es en la tabla variable, en donde la condición de ensamble rige sobre el nombre, el cual no es su clave primaria. Sin embargo esta tabla es muy chica (posee 8 registros y crece rara vez) así que no hace falta crear un índice para esto; sería excesivo puesto que el índice podría ocupar una porción de memoria más grande que la tabla en sí, sin mencionar que la velocidad no aumentaría en absoluto. Si se utiliza el comando *EXPLAIN ANALYZE* previo a la consulta se puede ver que se realizan dos escaneos completos sobre las condiciones de filtrado en la cláusula *WHERE*. Se procede entonces a crear un índice sobre el campo *valor* en la tabla *medicion* y cambiando la consulta puntual de un año extrayendo una parte de la fecha por un rango, tal como se hizo en consultas anteriores, con el fin de reutilizar un índice creado sobre este campo. La consulta final y optimizada puede verse a continuación, junto con la creación del índice faltante:

```
CREATE INDEX ix_valor_medicion ON medicion(valor);

SELECT D.*
FROM departamento D
INNER JOIN variable V ON(v.nombre = 'datos')
INNER JOIN medicion_departamento M ON (D.id_departamento = M.id_departamento AND
V.id_variable = M.id_variable)
INNER JOIN medicion X ON (X.id_departamento = M.id_departamento AND X.id_variable
= M.id_variable)
WHERE (
	(X.valor > 90) AND
	(X.fecha >= '2010-05-01'::date AND X.fecha < '2010-06-01'::date)
);
```

Por otro lado, el enunciado solicita que se retornen los datos de los departamentos que registran al menos una medición superando un valor umbral, con lo cual la consulta puede ser optimizada utilizando el operador *EXISTS*, el cual se utiliza justamente en estos casos.

```
SELECT D.*
FROM departamento D
WHERE EXISTS (
	SELECT M.*
	FROM variable V
	INNER JOIN medicion M ON (M.id_departamento = D.id_departamento AND
V.id_variable = M.id_variable)
	WHERE (
		(M.fecha >= '2010-05-01'::date AND M.fecha < '2010-06-01'::date) AND
		(M.valor > 90) AND
		(V.nombre = 'datos')
	)
```

```
        LIMIT 1
);
```

MongoDB

Este motor no posee el stage *$exists*. No obstante, posee *$in* pero es ineficiente y su uso no es recomendado. Para resolver esta narrativa se utilizó *$lookup* y filtros mediante stages *$match*.

En primer instancia, la consulta que resuelve la problemática dada es la siguiente:

```
var variable = db.variable.findOne({"nombre":"datos"});
var id_var = variable["_id"];

db.medicion.aggregate([
    {$match:
        {
            "id_variable": id_var,
            "fecha": {$gte: ISODate("2010-05-01T03:00:00.0Z"), $lt:
ISODate("2010-06-01T03:00:00.0Z")},
            "valor" : {$gt:90}
        }
     },
     {$lookup:
        {
            from: "departamento",
            localField: "id_departamento",
            foreignField: "_id",
            as: "departamento"
        }
     },
     {$unwind: "$departamento"},
     {$lookup:
        {
            from: "edificio",
            localField: "departamento.id_edificio",
            foreignField: "_id",
            as: "edificio"
        }
     },
     {$unwind:"$edificio"},
     {$project:
         {
            "pais": "$edificio.pais",
            "provincia": "$edificio.provincia",
            "ciudad": "$edificio.ciudad",
            "direccion": "$edificio.direccion",
            "nombre_depto": "$departamento.nombre_depto"
         }
     }
]);
```

En el stage $lookup se configuran cuatro campos: from, localField, foreignField y as como la colección con la cual se desea ensamblar, el campo seleccionado para la condición de ensamble según la colección obtenida del stage anterior (la colección local), el campo para la condición de ensamble de la colección a ensamblar y finalmente el nombre que va a poseer la estructura ensamblada al finalizar el stage. Para que esta consulta se ejecute más rápidamente es necesario crear un índice sobre el campo valor en la colección medicion ya que en este caso el filtro se encuentra en la primer parte del pipeline. La creación del índice se concreta con el siguiente comando:

```
db.medicion.createIndex({valor:1},{name:"ix_valor_medicion"});
```

Cabe destacar que el identificador de la variable fue buscado individualmente en lugar de ensamblarlo utilizando *$lookup* ya que de esta forma se realizaba la consulta en un tiempo menor, aumentando el rendimiento. Esto es posible ya que Mongo utiliza Javascript reducido, pudiendo almacenar documentos en variables para ser utilizadas posteriormente. Aquí se utilizó *findOne* con la condición de que el nombre sea igual a *datos*. Por otro lado, la agregación se hizo sobre la colección *medicion* ya que si esta estuviera en los stages de ensambles (*$lookup*), sólo sería posible referenciar a una parte de su clave identificadora; la clave de la colección es <*id_variable, id_departamento, fecha*>, pero *$lookup* sólo permite especificar un sólo campo foráneo. Esto implica que se deba elegir un campo y en el próximo stage se filtren los demás, lo cual lo haría ineficiente. De esta forma, llevando a cabo la agregación sobre la colección “conflictiva” se solucione el problema.

Esta consulta no óptima y puede mejorarse agregando un stage de agrupamiento, teniendo en cuenta un acumulador y sentido común:

```
var variable = db.variable.findOne({"nombre":"datos"});
var id_var = variable["_id"];

db.medicion.aggregate([
    {$match:
        {
            "id_variable": id_var,
            "fecha": {$gte: ISODate("2010-05-01T03:00:00.0Z"), $lt:
ISODate("2010-06-01T03:00:00.0Z")}
        }
     },
     {$project:
         {
            "id_departamento": "$id_departamento",
            "valor": "$valor"
         }
     },
     {$group:
        {
```

```
            _id: "$id_departamento",
            valor: {$max:"$valor"}
        }
    },
    {$match:
        {
            "valor" : {$gt:90}
        }
    },
    {$lookup:
        {
            from: "departamento",
            localField: "_id",
            foreignField: "_id",
            as: "departamento"
        }
    },
    {$unwind: "$departamento"},
    {$lookup:
        {
            from: "edificio",
            localField: "departamento.id_edificio",
            foreignField: "_id",
            as: "edificio"
        }
    },
    {$unwind:"$edificio"},
    {$project:
        {
            "pais": "$edificio.pais",
            "provincia": "$edificio.provincia",
            "ciudad": "$edificio.ciudad",
            "direccion": "$edificio.direccion",
            "nombre_depto": "$departamento.nombre_depto"
        }
    }
]);
```

El primer stage, *$match* es similar al anterior, sólo que ya no se filtra por valor. Luego se aplica *$project* para proyectar las columnas necesarias y se procede a agrupar por departamento y tomar el máximo valor relativo al consumo de datos del mismo departamento en el stage *$group*. Esto es así ya que si el máximo registrado no supera los 90GB entonces ningún otro registro del mismo departamento lo hará. En el próximo paso se filtran los que no superen los 90GB utilizando *$match* aunque en este caso Mongo ya no utiliza índices porque no está al principio del pipeline.

5.1.3.2 Consulta 7

Obtener el consumo total de datos de internet de todos los edificios registrados en estados unidos.

PostgreSQL

Esta consulta no puede ser optimizada por índices. Esto es así ya que la condición de filtrado se realiza sobre la tabla *edificio*, la cual es de las más chicas en lo que respecta a las filas.

```
SELECT SUM(M.valor)
FROM departamento D
INNER JOIN variable V ON (V.nombre = 'electricidad')
INNER JOIN medicion_departamento X ON (X.id_variable = V.id_variable AND
X.id_departamento = D.id_departamento)
INNER JOIN edificio E ON (E.id_edificio = D.id_edificio)
INNER JOIN medicion M ON (M.id_variable = V.id_variable AND M.id_departamento =
X.id_departamento)
WHERE (E.pais = 'United States');
```

En este caso, también se ensambla desde las tablas más chicas a las más grandes, con la excepción de la tabla *departamento*, la cual se la contempla antes que la tabla *variable*. Esto se hace por la misma razón que la consulta anterior; para ensamblar la tabla *departamento*, se la utiliza directamente desde *FROM* o bien se la ensambla luego de *medicion_departamento*, pero *medicion_departamento* es más grande que estas dos.

MongoDB

En esta consulta se establece un stage *$group* con la clave de agrupamiento nula, puesto que lo único que nos interesa obtener es la suma de todos los consumos individuales, sin agrupar otra información. Esta es la única forma de que Mongo permite llevar a cabo este tipo de operaciones; haciendo que el agrupador contenga al menos un campo dentro de *_id* o bien que este sea nulo.

```
var variable = db.variable.findOne({"nombre":"electricidad"});
var id_var = variable["_id"];
db.medicion.aggregate([
    {$match:
        {
            "id_variable": id_var
        }
     },
     {$group:
        {
            _id: "$id_departamento",
            valor: {$sum:"$valor"}
        }
     },
     {$lookup:
        {
            from: "departamento",
            localField: "_id",
            foreignField: "_id",
```

```
            as: "departamento"
         }
      },
      {$unwind: "$departamento"},
      {$lookup:
         {
            from: "edificio",
            localField: "departamento.id_edificio",
            foreignField: "_id",
            as: "edificio"
         }
      },
      {$unwind:"$edificio"},
      {$match:
          {
             "edificio.pais":"United States"
          }
      },
      {$group:
          {
             _id:null,
             "consumoTotal": {$sum:"$valor"}
          }
      }
]);
```

5.1.3.3 Consulta 8

Se desea obtener la sumatoria de los excedentes de gas en los registros de mediciones de cada departamento desde el año 2014 al 2016 inclusive.

En esta consulta se deben realizar ensambles y funciones de agregación, así como filtrados por fechas, de forma similar a las consultas anteriores y reutilizando los índices ya definidos.

PostgreSQL

Esta consulta es similar a las anteriores, sólo que realiza una resta dentro de la función de agregación suma para sumar los excedentes en cada registro que exceda el valor máximo permitido de la variable gas en sus mediciones efectivas. En las condiciones se opera de forma tal de poder reutilizar los índices de *valor* y *fecha* en la tabla *medición*.

```
SELECT M.id_departamento, SUM(M.valor - V.valmax)
FROM variable V
INNER JOIN medicion M ON (M.id_variable = V.id_variable)
WHERE (
       (M.valor > V.valmax) AND
       (M.fecha >= '2014-01-01'::date AND M.fecha < '2017-01-01'::date) AND
       (V.nombre = 'gas')
)
GROUP BY M.id_departamento;
```

MongoDB

De forma similar a los casos previos de ensamble, primero se busca el identificador de la variable y su valor máximo en la colección *variable* y luego se procede a una segunda consulta mediante *aggregate()*, como se puede ver a continuación:

```
var variable = db.variable.findOne({nombre:"gas"});
var valmax = variable["valmax"];
var id_variable = variable["_id"];

db.medicion.aggregate([
    {$match:
        {
            "fecha": {$gte: ISODate("2014-01-01T03:00:00.0Z"), $lt:
ISODate("2017-01-01T03:00:00.0Z")},
            "id_variable": id_variable,
            "valor": {$gt: valmax}
        }
     },
     {$project:
         {
            "id_departamento": "$id_departamento",
            resta: {$subtract: ["$valor", valmax]}
         }
     },
     {$group:
         {
            _id: {"id_departamento":"$id_departamento"},
            excedente: {$sum: "$resta"}
         }
     }
]);
```

Como el primer stage es *$match*, es posible que el planificador utilice los índices creados sobre *valor* y *fecha* de la colección *medicion.* En el primer paso se filtran sólo los documentos que excedan el valor máximo permitido en las mediciones de gas en el rango de fecha solicitado. Existe un stage intermedio, en donde se proyectan los excedentes junto con los identificadores de los departamentos. Esto es así, ya que no se puede operar con la resta dentro de la función de agregación, entonces en el paso anterior hay que modificar la información realizando esta resta. En el tercer stage se agrupa por departamento y se suman sus excedentes proyectados.

5.2 Consultas de Actualización

En esta sección se propondrán dos consultas de actualización y se discutirán las implementaciones. Todas las modificaciones serán acumulativas y también serán visibles en las secciones siguientes.

Consulta 1

El equipo de inspección verificó las instalaciones en los edificios logrando identificar errores en los sensores de electricidad, los cuales medían 10W más del escenario real. Se desea modificar todos los registros de medición de electricidad restando 10W en cada caso.

PostgreSQL

En esta caso existirá una subconsulta para obtener primero el identificador de la variable cuyo nombre sea 'electricidad' para luego restar el valor 10 (el cual representa los watts) en cada registro perteneciente a las mediciones efectivas de la electricidad en la tabla *medicion*. La consulta de PostgreSQL que resuelve la problemática dada se muestra a continuación:

```
UPDATE medicion
SET valor = valor - 10
WHERE id_variable = (
	SELECT V.id_variable
	FROM variable V
	WHERE V.nombre = 'electricidad'
	LIMIT 1
);
```

Puede verse que en la condición hay una igualdad entre *id_variable*. Esto es así ya que se sabe de antemano que la subconsulta no devolverá más de un valor por *LIMIT 1*; si esta restricción no existiese la consulta podría fallar, ya que la comparación con '=' no puede ser entre un valor y un *set* ya que para ello se utiliza *IN* o *EXISTS*.

MongoDB

En el caso de Mongo, la consulta se hace utilizando la función *update*. La selección del identificador de la variable con nombre 'electricidad' se realiza consultado el valor del identificador extraído mediante una consulta por el nombre en la colección *variable'*. Como se extrae el valor utilizando *findOne*, sabemos que sólo obtendremos un valor.

```
var variable = db.variable.findOne({nombre:"electricidad"});
var id_variable = variable["_id"];

db.medicion.update(
	{"id_variable": id_variable},
	{$inc:{"valor":-10}},
	{multi:true}
);
```

El primer parámetro de *update* representa la condición de selección. El segundo pafametro utiliza *$inc* para incrementar el valor de los documentos matcheados en *-10*; como no hay resta, se utiliza el incremento de un número negativo. En las actualizaciones el motor nos fuerza a utilizar estos tipos de modificadores; los más utilizados son *$inc* y *$set*, aunque

existen muchos otros. Finalmente, el último parámetros establece un flag en *true*, el cual solicita que se modifiquen todos los documentos matcheados; si no se lo establece o bien es *false* entonces sólo modificará al primero y terminará la ejecución.

Consulta 2

Se realizó un cambio de equipos de sensores de humo, humedad e iluminación, los cuales registraban valores erróneos en el departamento con identificador 26. La compañía decidió que estos valores no son útiles para el análisis de información, con lo cual decidió setearlos como cero.

PostgreSQL

A diferencia de la consulta anterior, la subconsulta devuelve más de un valor, con lo cual se utiliza *IN*. La consulta en SQL que resuelve el pedido es la siguiente:

```
UPDATE medicion
SET valor = 0
WHERE (
	(id_departamento = 26) AND
	(id_variable IN
		(SELECT V.id_variable
		FROM variable V
		WHERE V.nombre IN ('humo', 'humedad', 'iluminacion')
		)
	)
);
```

MongoDB

En primer lugar, es necesario obtener los identificadores de las variables a utilizar. A diferencia de la consulta anterior se debe tener en cuenta tres variables. Por ello se buscan los identificadores de las variables ‘humo’, ‘humedad’ e ‘ilumunacion’ en la colección *variable* y se recorren los resultados para insertarlos en un arreglo para procesarlos en la consulta de actualización:

```
var docs = db.variable.find({$or:[{nombre:"humo"}, {nombre:"humedad"},
{nombre:"iluminacion"}]});
var ids = [];

//se insertan los valores en el arreglo ‘ids’
docs.forEach(function(doc){
	ids.push(doc["_id"]);
});

db.medicion.update(
	{$and:[{"id_departamento":26},{"id_variable": {$in:ids}}]},
	{$set:{"valor":0}},
	{multi:true}
);
```

Como puede verse, se utiliza el operador *$and* en la condición, el cual es un y-lógico. En primer lugar el identificador del departamento debe ser 26 y por otro lado, el identificador de la variable medida debe estar dentro del arreglo *ids*, utilizando *$in*. En segundo lugar, se emplea *$set* para modificar el valor del campo por otro valor fijo en los documentos filtrados.

5.3 Consultas de Eliminación

En esta sección se discutirán las implementaciones en cada motor de dos consultas de eliminación sobre el esquema/diseño en cada motor. Las modificaciones serán acumulativas.

Consulta 1

El sistema se encuentra en etapa de migración pero los los desarrolladores decidieron copiar la información de las mediciones desde 2010 en adelante y conservar la información previa a ese año. Como los datos posteriores al 2010 ya se salvaron en otra base de datos, es necesario eliminarlos.

PostgreSQL

La eliminación de los datos posteriores al año 2010 en la tabla *medicion* se realiza con la siguiente consulta:

```
DELETE FROM medicion
WHERE (fecha >= '2010-01-01'::date);
```

MongoDB

En MongoDB se utiliza la función *delete*, en donde sólo se explicita una condición de filtrado. En este caso, a diferencia de *update()*, no es necesario establecer el flag *multi:true* ya que por defecto elimina múltiples documentos. La primer consulta puede verse a continuación:

```
db.medicion.remove({"fecha": {$gte: ISODate("2010-01-01T03:00:00.0Z")}});
```

Sin embargo, esta consulta toma horas y a medida que los datos crecen, puede tomar días. Eventualmente, el servidor no responde sobre la colección ya que durante su ejecución no se realiza un *lock* sobre ella por completo. lo cual es inadmisible en sistemas en tiempo real. Esto se da, debido a que las eliminaciones de grandes volúmenes de datos típicamente se realizan sobre estructuras anidadas pero en este caso la colección no podría estar agrupada por las fechas, debido a que cada medición efectiva se concreta en términos de año, mes, día, hora, minuto y segundo; particular en cada variable a medir y el departamento. Además, si la mejora de esta operación implica el cambio en la organización, entonces podría ocurrir que las consultas de selección se vieran afectadas perdiendo performance, lo cual no es deseado ya que esta eliminación es un caso excepcional que ocurrirá sólo una vez. Dadas las circunstancias, una opción con un enfoque distinto es el de copiar los datos de la colección excepto los que se deseen eliminar, luego eliminar la colección anterior por completo, copiar

los índices y renombrar la nueva colección. Esto es terriblemente ineficiente y es poco recomendado, ya que es una mala práctica. No obstante, este es un caso particular y sólo ocurrirá una sola vez pudiendo realizarse en paralelo permitiendo entonces que los demás clientes puedan seguir accediendo a la primer colección mientras dura este proceso. La solución es la siguiente:

```
//Copiamos los elementos, excepto los que sean mayores al año 2010 (los que
queremos eliminar).
db.medicion.find({"fecha": {$lt:
ISODate("2010-01-01T03:00:00.0Z")}}).forEach(function(doc){
    db.medicionAux.insert(doc);
});

//Se crean los mismos índices en la nueva colección
db.medicion.getIndexes().forEach(function(ix){
    db.medicionAux.ensureIndex(ix.key);
});

//Se renombra según la colección anterior, y la colección anterior se elimina.
db.medicionAux.renameCollection(db.medicion.getName(), {dropTarget: true});
```

Esta consulta toma minutos, lo cual sigue siendo malo, pero mejore notablemente la eliminación por *remove()*. Se recalca que esta utilización es muy especial, sólo para casos como este; generalmente las eliminaciones son puntuales o en rangos pequeños, y no de esta forma.

Consulta 2

Se desea eliminar toda la información de las mediciones efectivas relativas a los departamentos en el rango 5.000 a 16.000.

PostgreSQL

A continuación se muestra la consulta que resuelve el escenario propuesto:

```
DELETE FROM medicion
WHERE id_departamento BETWEEN 5000 AND 16000;
```

MongoDB

En el caso de mongo se aplica *remove()*:

```
db.medicion.remove({id_departamento:{$gte:5000, $lte:16000}});
```

Esta consulta también consume mucho tiempo, pero no tanto como la eliminación anterior, con lo cual no es necesario copiar la colección y renombrarla.

6. Métricas

En esta sección se evalúan los resultados en ambas bases de datos, realizando un análisis en cada caso. Para llevar a cabo las pruebas se utilizó PostgreSQL 9.5.4 y MongoDB 3.4.4 bajo el sistema operativo Ubuntu Xenial 16.04.1 LTS sobre un equipo con procesador Intel Celeron 1007U dual core con 1.5GHz y memoria RAM de 4GB.

6.1 Consultas de Inserción

Las inserciones se corresponden al total de datos sobre todas las tablas/colecciones utilizadas para llevar a cabo el resto de las consultas. La suma total de registros a almacenar son 8.000.000, distribuidos entre las distintas tablas/colecciones, a saber: 10 constructoras, 100 edificios, 50.000 departamentos, 8 variables, 400.000 registros de medición de variables en departamentos (*medicion_departamento*) y 8.000.000 mediciones efectivas (*medicion*). En PostgreSQL las inserciones tomaron 1241.09 segundos y en MongoDB 340.76 segundos, con lo cual MongoDB presenta una aceleración del 3.64 en términos de velocidad de escritura. En este caso, ambos motores tienen índices creados sobre las tablas/colecciones ya que se intenta simular lo que ocurriría en un sistema real. Estos resultados pueden verse en el siguiente gráfico [figura 5].

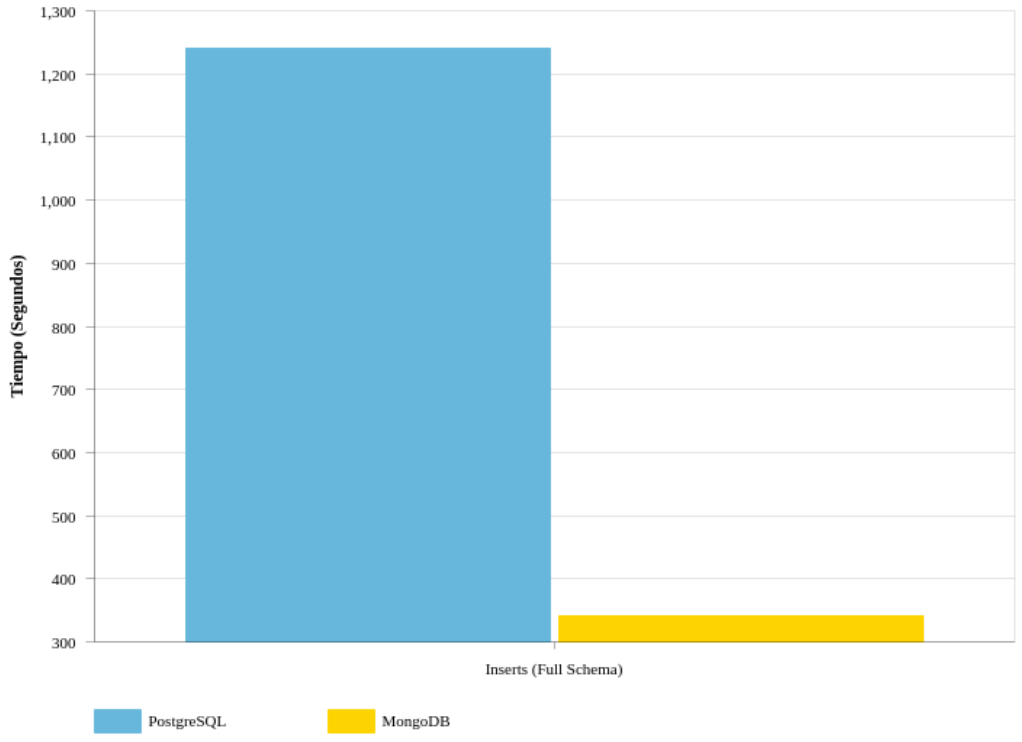

Figura 5: *gráfico de resultados de las inserciones.*

La escritura de grandes volúmenes de información ha resultado a favor de MongoDB por gran diferencia. Esto se debe a que Mongo está preparada para recibir tal cantidad de información en cortos lapsos de tiempo y reaccionar rápidamente, ya que se la suele utilizar en aplicaciones de tiempo real.

6.2 Consultas de Selección

En esta sección se discutirán los resultados de las consultas de selección contrastando los resultados en ambos motores.

6.2.1 Selecciones con Agrupadores y Funciones de Agregación

Las funciones de agregación y agrupamiento arrojaron los siguientes resultados [tabla 2].

Consulta	PostgreSQL	MongoDB
Consulta 1	6.77 segundos	11.16 segundos
Consulta 2	0.19 segundos	0.17 segundos
Consulta 3	7.95 segundos	12.21 segundos

Tabla 1: *valores medios de los tiempos de las consultas de selección con agrupadores y funciones de agregación.*

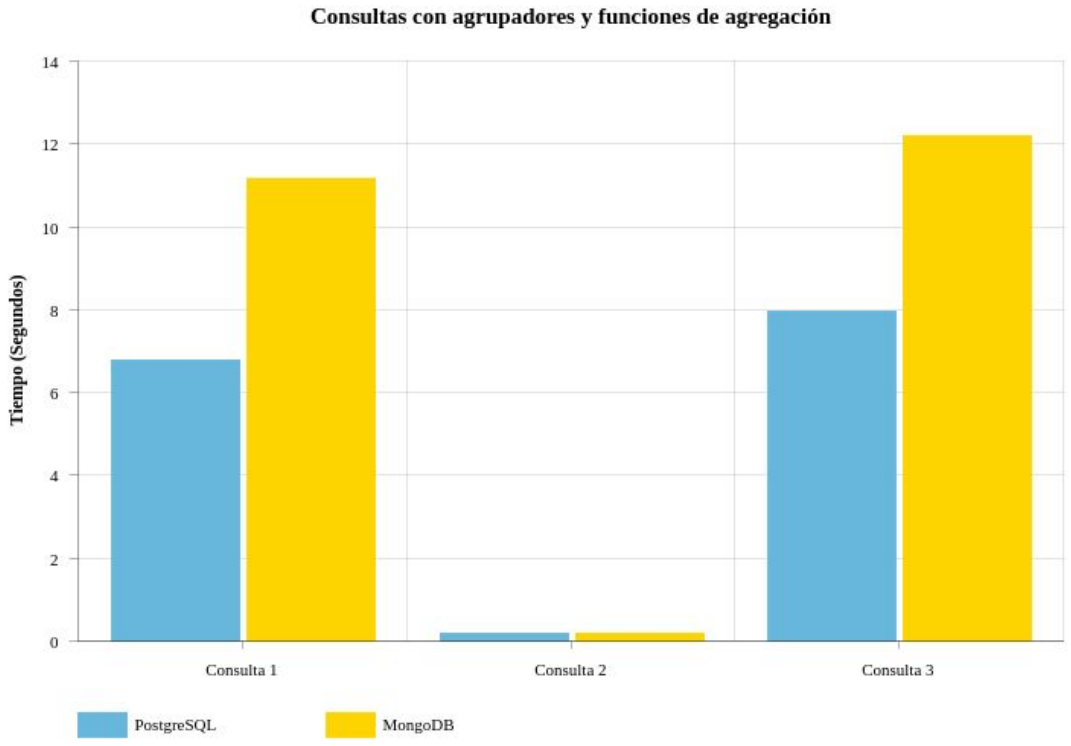

Figura 6: *gráfico de resultados de las consultas de selección con agrupadores y funciones de agregación.*

Como puede observarse en el gráfico de resultados [figura 6], los resultados favorecen a MongoDB en dos de las tres consultas. La consulta 2 tomó poco tiempo debido a que se llevó a cabo sobre un conjunto de datos pequeño, pero la consulta 1 y la consulta 2 consumen un tiempo considerable en ambas bases de datos. Si MongoDB tuviese sharding, las consultas 1 y 3 demoraría mucho menos que PostgreSQL, teniendo en cuenta que la clave de distribución sería *id_variable*, sin embargo, en un sólo nodo de ejecución los resultados son mejores en la base de datos relacional.

6.2.2 Selecciones sin Repeticiones

Los resultados de las consultas sin repeticiones pueden verse a continuación [tabla 2].

Consulta	**PostgreSQL**	**MongoDB**
Consulta 4	1.42 segundos	1.60 segundos
Consulta 5	32.84 segundos	13.45 segundos

Tabla 2: *valores medios de los tiempos de las consultas de selección sin repeticiones.*

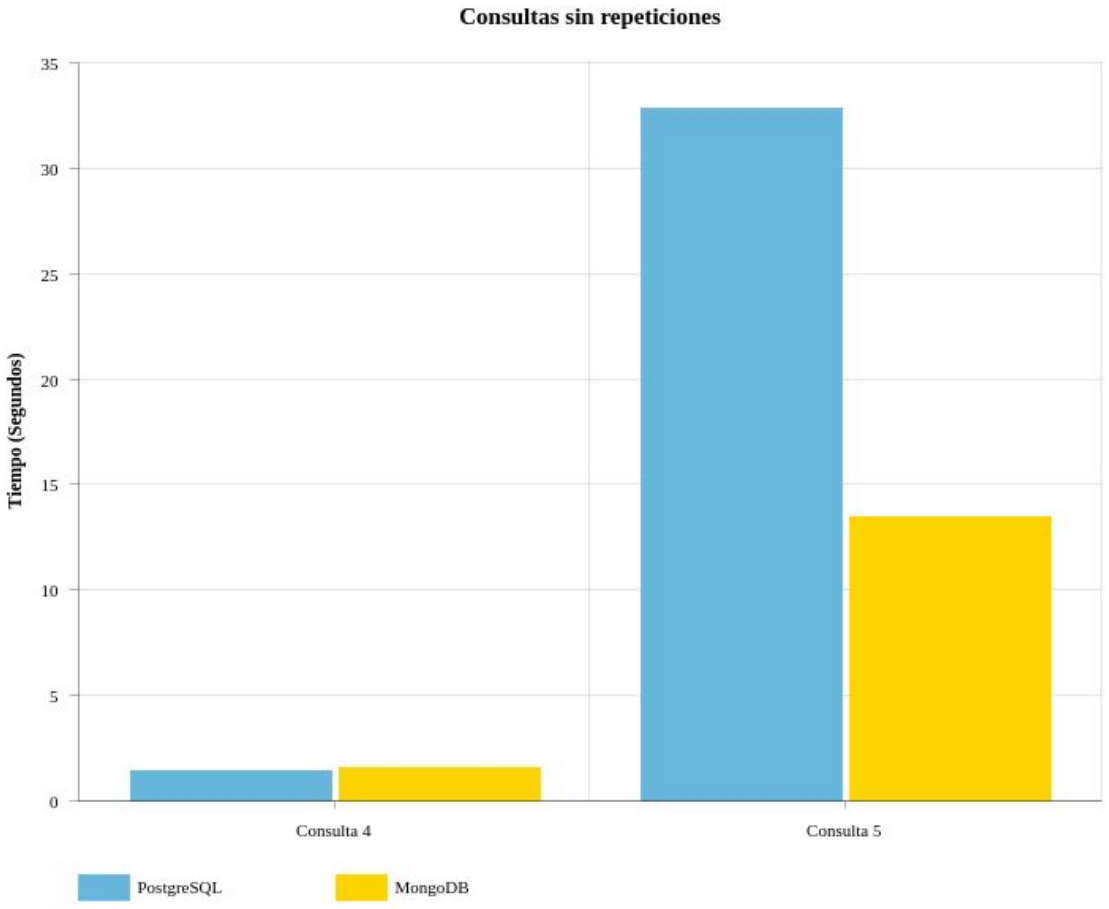

Gráfico 7: *gráfico de resultados de las consultas de selección sin repetición.*

La eliminación de repetidos en las consultas se comportaron mejor en MongoDB, tal como puede visualizarse en el gráfico anterior [figura 7]. En la consulta 4, la diferencia fue ínfima a favor de PotgreSQL, pero en la consulta 5 hay una diferencia de varios segundos. En esta última, no existe condición de filtrado, por lo que la cantidad de datos devueltos son millones. Además, se debe extraer del campo *fecha* el año y el mes para eliminar las repeticiones respecto a éstos, lo cual agrega un overhead a la consulta.

6.2.3 Selecciones con Ensambles

Las condiciones con ensambles fueron rápidas en ambo motores, al tratar con tal volumen de datos. Los resultados medios arrojados por las pruebas pueden verse a continuación [tabla 3].

Consulta	PostgreSQL	MongoDB
Consulta 6	0.55 segundos	0.29 segundos
Consulta 7	0.33 segundos	0.25 segundos

Consulta 8	1.50 segundos	12.59 segundos

Tabla 3: *valores medios de los tiempos de las consultas de selección sin repeticiones.*

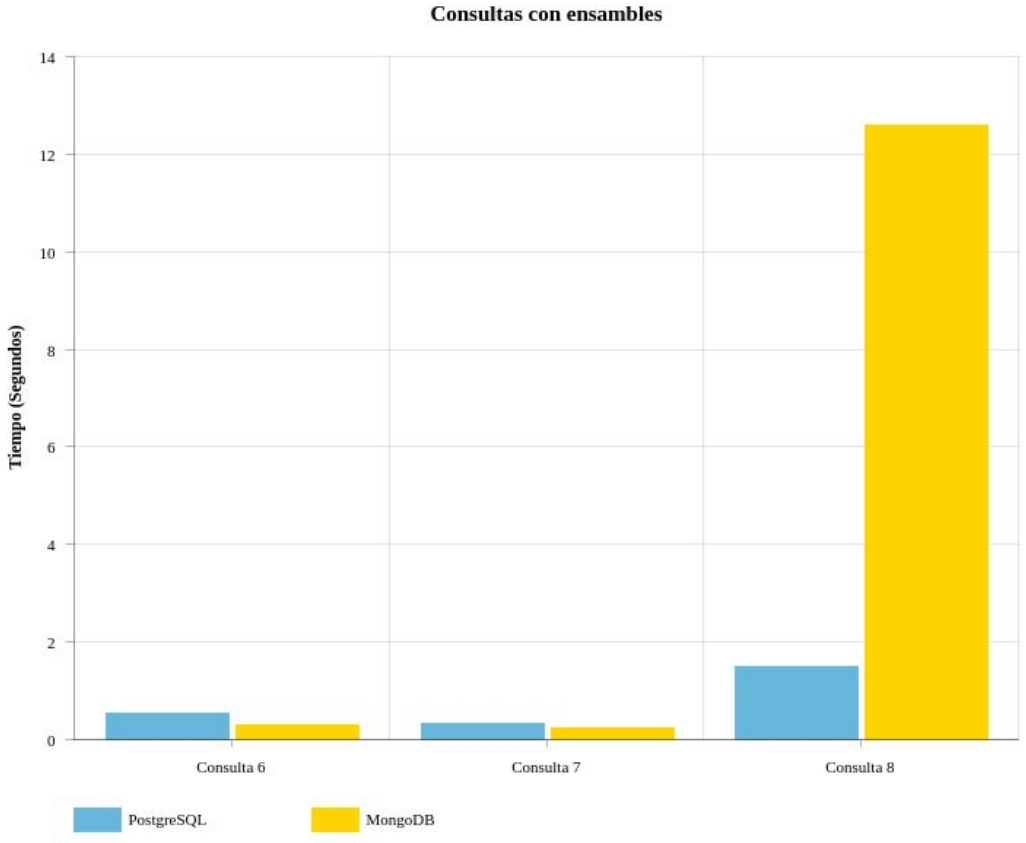

Gráfico 8: *gráfico de resultados de las consultas de selección con ensambles.*

Como puede verse en el gráfico de tiempo [figura 8], las consultas 6 y 7 son mejores en MongoDB que en PostgreSQL. Por otro lado, la consulta 8 pierde en el caso de PostgreSQL debido a que se utilizan funciones de agregación con stages intermedios para modificación de los datos filtrados; en este caso, una resta para obtener los excedentes. Es importante destacar que los stages *$lookup* son muy recientes en MongoDB y a pesar de estar en versiones tempranas se comporta muy bien y a medida que el volumen de datos crezca, obtendrá mejores tiempos que PostgreSQL.

6.3 Consultas de Actualización

En este caso se probaron dos consultas de actualización, cuyos resultados pueden verse a continuación [tabla 4]. La consulta 1 modifica 1.000.000 de registros, mientras que la segunda consulta modifica 60 registros.

Consulta	**PostgreSQL**	**MongoDB**

Consulta 1	22.6 segundos	150.279 segundos
Consulta 2	0.144 segundos	0.951 segundos

Tabla 4: *valores medios de los tiempos de las consultas de selección sin repeticiones.*

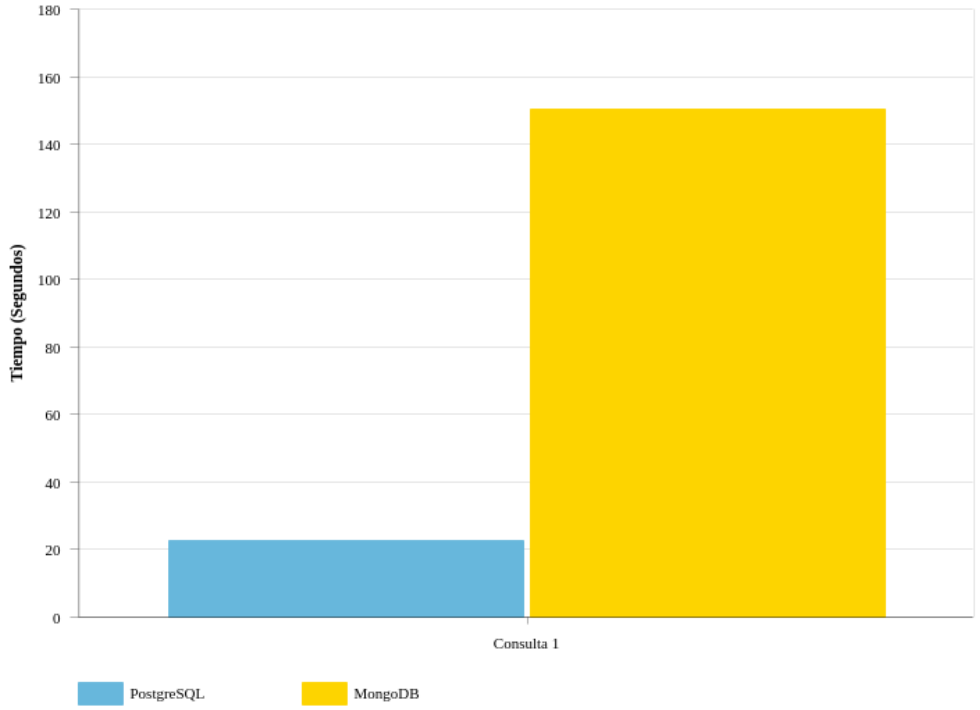

Gráfico 9: *gráfico de resultados de la primer consulta de actualización.*

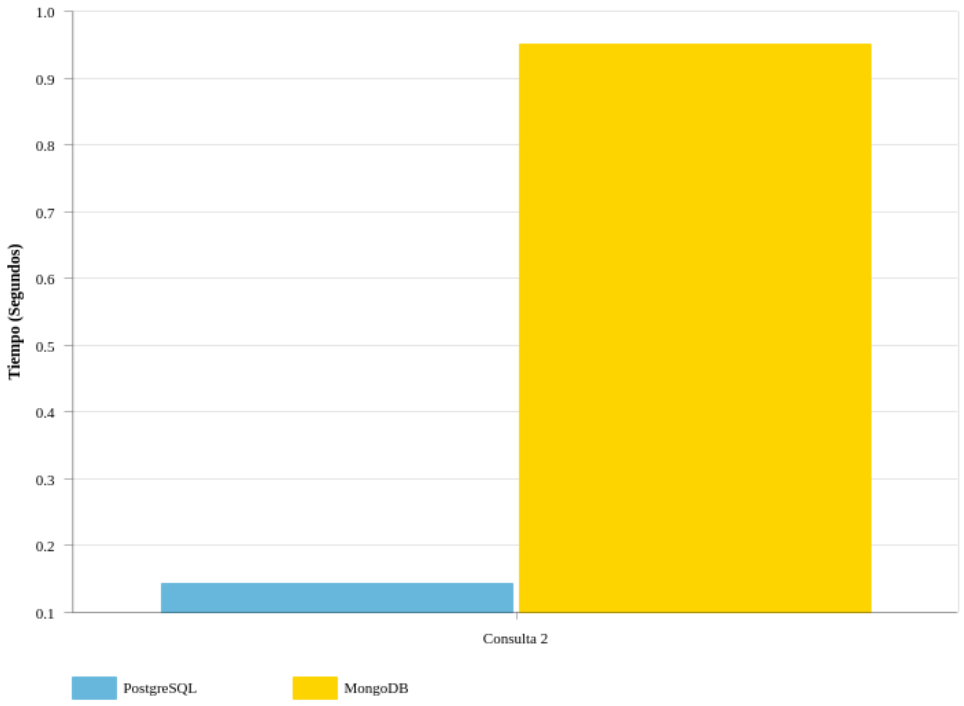

Figura 10: *gráfico de resultados de la segunda consulta de actualización.*

Como puede verse, mongo resultó ser más eficiente en ambas consultas por gran diferencia, debido a que presentó una aceleración del 6.64 y 6.6 en la consulta 1 y 2, respectivamente en contraste a MongoDB.

6.4 Consultas de Eliminación

La primer consulta toma 14.5 segundos en PostgreSQL y en el caso de MongoDB, la consulta con la copia de la colección filtrando los datos eliminados toma 20.5 minutos. Por otro lado, la segunda consulta toma 7.9 segundos y 25.8 minutos en MongoDB. Puede verse entonces que la diferencia es abismal y se debe en parte a las formas internas que tiene Mongo para eliminar la información y los accesos a disco; las operaciones de eliminación toman demasiado tiempo a diferencia de PostgreSQL.

Los gráficos en esta sección no son representativos, ya que en escala de minutos el tiempo tomado de las consultas en PostgreSQL no serían vistos.

7. Conclusión

Este proyecto integrador permitió explorar dos motores de almacenamiento sumamente populares de distinta naturaleza. Particularmente, mi percepción sobre las bases de datos NoSQL es que serían más rápidas que las bases de datos relacionales. No sólo esto no es cierto, sino que también depende mucho del contexto en el cual se lo aplique. Como pudo verse en los resultados, PostgreSQL brindó una mayor performance que la utilización de MongoDB en la mayoría de las métricas. Si se requieren consultas rápidas en un único nodo de procesamiento es conveniente utilizar PostgreSQL, pero si se está en el contexto distribuído la mejor opción sería MongoDB. Un posible trabajo a futuro sería evaluar estos

dos motores con la misma organización de la información utilizando sharding para evaluar su rendimiento. Asimismo, y a partir del análisis en las secciones de la organización de la información y los resultados arrojados se ve que el diseño de entidades en Mongo no fue ideal, pero se descartaron otras opciones justamente por limitaciones del propio paradigma. Al tratarse de un caso en el cual el anidamiento no resultaba conveniente, se intentó definir un diseño totalmente normalizado similar al del esquema relacional, lo cual en este caso particular fue útil, pero lo recomendable es la denormalización siempre que sea posible. Los casos en los que el anidamiento es conveniente es cuando no se realizan consultas teniendo en cuenta los documentos anidados, es decir, las condiciones se aplican sobre los documentos de la colección y no en estructuras embebidas. Por ejemplo, una foto con comentarios y "me gusta" son ideales para un esquema anidado, ya que las búsquedas se realizan sobre las fotos y no sobre la información en la información anidada. La elección de esta organización se vio reflejada en los distintos síntomas respecto a la performance, ya que las eliminaciones y ciertas búsquedas de agrupamiento fueron lentas. No obstante, el dominio no permitió una organización diferente lo cual reafirma que la utilización de una base de datos u otra depende en su totalidad de los atributos de calidad del sistema.

A modo de opinión, la utilización de MongoDB fue positiva, dado que me permitió conocer nuevos paradigmas dentro de las bases de datos que hasta el momento me eran desconocidas. Por otro lado, fue muy valioso comprender que no se trata de qué base de datos reemplaza a otra sino de saber en qué contexto se utiliza cada una de ellas o bien cuándo pueden coexistir en el mismo sistema dando una visión mucho más amplia utilizando lo mejor de ambos motores. Otro tema importante a cubrir fue la migración de datos de PostgreSQL a MongoDB, lo cual fue demasiado complicado dado que no hay herramientas para hacerlo o bien no son gratuitas, lo cual implicó un programa escrito en Java. También se incluyen scripts y la complejidad de aprender cómo realizar las consultas con utilidades experimentales en MongoDB poco exploradas en la web.

8. Referencias

[1] Rodríguez, A. (2008). Sistemas SCADA. 2 ed. Barcelona: Editorial Marcombo.
[2] Harizopoulos, S., Abadi, D. J., Madden, S., & Stonebraker, M. (2008, June). OLTP through the looking glass, and what we found there. In Proceedings of the 2008 ACM SIGMOD international conference on Management of data (pp. 981-992). ACM.