BASE DE DATOS NO RELACIONALES

Laboratorio 1

Author
Santiago Calvo - 5.055.578-2
Emiliano de Sejas - 5.088.428-8

Contents

1 Google Collab **3**

2 Queries **3**

2.1 Costos de acceso 3

2.1.1 Costos customers_list 4

2.1.2 Costos film_list 5

2.1.3 Costos sales_by_store 5

2.1.4 Costos sales_by_film_category 6

3 Optimizaciones **6**

4 Costos con optimizaciones **8**

4.1 Costos de customers_list 8

4.2 Costos de film_list 8

4.3 Costos de sales_by_store 9

4.4 Costos sales_by_film_category 10

5 DD1 vs DD2 **11**

6 Conclusión **12**

1 Google Collab

Las queries realizadas pueden encontrar aquí, como un ambiente en google collab.
https://colab.research.google.com/github/sncalvo/improved-broccoli/blob/main/tarea.ipynb

2 Queries

Todas las queries pueden encontrarse en el Jupyter Notebook asociado. En esta sección realizaremos un estudio de los costos de accesos antes y después de las optimizaciones realizadas.

2.1 Costos de acceso

Para realizar el análisis de los costos haremos la descomposición de las agregaciones en cada etapa (*stage*), simplificando el desarrollo.

Tomamos para los tiempos de acceso las siguientes heurísticas:

1. El costo por acceder a un documento por su _id o clave primaria o indice único es 1 unidad.
2. El costo por acceder a un documento en una colección sin ningún índice se puede asumir que es en promedio la mitad del tamaño de la colección.
3. El costo por acceder a cada atributo simple dentro de un documento es 0.01 (por ejemplo se accede al momento de proyectar o imprimir)
4. La distribución de las consultas en un día tipo es de esta forma:
   (a) customer_list=100 veces por día
   (b) film_list=80 veces por día
   (c) sales_by_store=400 veces por día
   (d) sales_by_film_category=500 veces por día

A su vez se realizaron algunas suposiciones educadas sobre los tiempos de ciertos accesos:

1. Los lookup que realizan pedidos que podrian terminar en multiples resultados se asumieron que tienen un costo de recorrer todos los registros, puesto que el motor no puede solamente comprar con el primer resultado.
2. Al realizar un lookup, hay que tomar en cuenta los costos de acceder a los atributos utilizados por la operación.
3. En cada caso se tomó el peor caso razonable en cuanto a número de registros retornados.

A su vez, todos los calculos se realizaron con los siguientes datos:

$$|customer| = 599$$
$$|address| = 603$$
$$|city| = 600$$
$$|country| = 109$$
$$|film\_category| = 1000$$
$$|film| = 1000$$
$$|category| = 16$$
$$|film\_actor| = 5462$$
$$|actor| = 200$$
$$|payment| = 16044$$
$$|rental| = 16044$$
$$|staff| = 2$$
$$|inventory| = 4581$$

En cada estudio de costo, se utilizó n para referirse al número de elementos en la tabla inicial.

Luego se utilizó las iniciales de cada tabla (o sus dos primeras iniciales en caso de ambigüedad) para referirse al máximo número de elementos de cada tabla.

2.1.1 Costos customers_list

Tomando n como el número de *customers*, cada stage realiza los lookups necesarios para pedir *address*, de *address* a *city* y luego de *city* a *country*.

Para cada uno de estos lookups se realiza la busqueda del registro necesario dentro de la tabla. Para esto es necesario acceder al atributo que contiene el indice de referncia, para luego realizar las comparaciones en las tablas objetivo, donde tambien debe obtenerse el id necesario.

El último paso necesario es la proyección donde solo se debe acceder a atributos para obtener el resultado final.

Esto se plantea de la siguiente manera:

$$
\begin{aligned}
|customers| &= n * 1 \\
|lookup\ address| &= n * (0.01 + a/2(1 + 0.01)) \\
|lookup\ city| &= (3 * 0.01 * n) + n * ci/2 * (1 + 0.01) \\
|lookup\ country| &= (3 * 0.01 * n) + n * co/2 * (1 + 0.01) \\
|set| &= (3 * 0.01 * 2) * n \\
|project| &= (2 * 0.01 + 2 * 0.02 * 5 + 0.01 + 0.02) * n \\
|total| &= \sum stages \\
&= n + n * (0.01 + a/2(1.01)) + (0.03 * n) + \\
&\quad n * ci/2 * (1.01) + (0.03 * n) + n * co/2 * (1.01) + (0.03 * 2) * n \\
|total| &= 3975959.31 \\
|total_e ach_d ay| &= 736280
\end{aligned}
$$

2.1.2 Costos film_list

De forma similar a *customers*, se deben buscar cada $film\_category$ y *category* utilizando lookups.

La diferencia con la query anterior surge al obtener $film\_actors$ donde los actores pueden ser multiples por cada $film$, por lo que tomamos como tiempo la cantidad máxima de $film\_actors$ (para poder obtener los k flim actores necesarios se deberá buscar sobre toda la tabla al no estar esta indexada).

El caso del lookup de *actors* es similar, aunque se deben comparar con multiples ids del resultado de $film\_actor$. Esto implica multiples accesos a atributos.

Luego proyectamos y obtenemos los resultados finales.

$$
\begin{aligned}
|films| &= n * 1 \\
|lookup\ film\_category| &= n * (0.01 + fc/2(1 + 0.01)) \\
|lookup\ category| &= (3 * 0.01 * n) + n * ca/2 * (1 + 0.01) \\
|lookup\ film\_actor| &= n * (0.01 + fa * (1 + 0.01)) \\
|lookup\ actor| &= (0.01 * n) + fa * (0.01 + 0.01 + a/2 * (1 + 0.01)) \\
|project| &= (0.01 * 5 + 0.01 * 3 + 0.01 + fa * 0.01 * 3) * n \\
|total| &= \sum stages \\
&= n + n * (0.01 + fc/2(1.01)) + (0.03 * n) + n * ca/2 * (1.01) + n * (0.01 + fa * (1.01) + \\
&\quad (0.01 * n) + fa * (0.02 + a/2 * (1.01)) + (0.09 + fa * 0.03) * n \\
|total| &= 6746481.24 \\
|total_e ach_d ay| &= 539718499.2
\end{aligned}
$$

2.1.3 Costos sales_by_store

Se realizan operaciones similares en este caso a las busquedas anteriores. El caso particular es el uso de un sort.

Para este se asumió un cost n veces el acceso de atributos necesarios para las comparaciones.

$$
\begin{aligned}
|stores| &= n * 1 \\
|lookup\ inventory| &= n * (0.01 + i * (1 + 0.01)) \\
|lookup\ rental| &= (0.01 * n) + i * (0.01 + 0.01 + r * (1 + 0.01)) \\
|lookup\ payment| &= (0.01 * n) + r * (0.01 + 0.01 + p/2 * (1 + 0.01)) \\
|lookup\ address| &= n * (0.01 + a/2(1 + 0.01)) \\
|lookup\ city| &= (3 * 0.01 * n) + n * ci/2 * (1 + 0.01) \\
|lookup\ country| &= (3 * 0.01 * n) + n * co/2 * (1 + 0.01) \\
|lookup\ staff| &= n * (0.01 + s/2(1 + 0.01)) \\
|set| &= n * (0.02 * 3 + 0.01) + p * (0.01 + 0.01) \\
|sort| &= n * 0.02 \\
|project| &= n * (0.04 * 2 + 0.02) \\
|total| &= n + n(0.01 + i * (1.01)) + (0.01 * n) + i * (0.02 + r * (1.01)) + (0.01 * n) + \\
&\quad r * (0.02 + p/2 * (1.01)) + n * (0.01 + a/2(1.01)) + (0.03 * n) + n * ci/2 * (1.01) + (0.03 * n) + \\
&\quad n * co/2 * (1.01) + n * (0.01 + s/2(1.01)) + n * (0.07) + p * (0.02) + n * 0.02 + n * (0.1) \\
|total| &= 204235467.03 \\
|total_{e}ach_{d}ay| &= 81694186811.99
\end{aligned}
$$

2.1.4 Costos sales_by_film_category

$$
\begin{aligned}
|category| &= n * 1 \\
|lookup\ film\_category| &= n * (0.01 + fc * (1 + 0.01)) \\
|lookup\ film| &= n * (0.01) + fc * (0.01 + f/2 * (1 + 0.01)) \\
|lookup\ inventory| &= n * (0.01) + fc * (0.01 + i * (1 + 0.01)) \\
|lookup\ rental| &= n * (0.01) + i * (0.01 + r * (1 + 0.01)) \\
|lookup\ payment| &= n * (0.01) + r * (0.01 + p/2 * (1 + 0.01)) \\
|project| &= n * (0.01) + (p * 0.01) \\
|total| &= n + n(0.01 + fc * 1.01) + n * (0.01) + fc * (0.01 + f/2 * (1 + 0.01)) + n * (0.01) + \\
&\quad fc * (0.01 + i * (1.01)) + \\
&\quad n * (0.01) + i * (0.01 + r * (1.01)) + n * 0.01 + r * (0.01 + p/2 * 1.01) + \\
&\quad n * (0.01) + (p * 0.01) \\
|total| &= 209372930.97 \\
|total_{e}ach_{d}ay| &= 104686465485
\end{aligned}
$$

3 Optimizaciones

Para reducir el costo de las consultas solicitadas, se modificaron algunas de las colecciones de DD1. El primer y más evidente cambio fue agregar índices a todos los atributos *id*

originados de la base relacional. Si bien se podría haber usado el *id* interno de *MongoDB*, decidimos no omitirlos ya que más tarde nos ayudarán a realizar comprobaciones sobre los resultados al compararlos con los datos originales.

En base a estas optimizaciones obtenemos la base DD2.

La siguiente optimización fue la de remover colecciones intermedias innecesarias y embeber aquellas colecciones que no tenían un volumen significativo de registros.

A continuación presentamos la lista de cambios que realizamos:

1. La colección *payment* fue removida y su contenido fue embebido en los registros de la colección *rental* correspondientes. La cantidad de payments asociados a un rental es 1 para todos los rentals en la base, dado a que es un cardinal muy bajo y no interesa acceder a los payments de forma independiente a los rentals (ninguna vista hace uso de estos por separado), justifica embeberlos en la colección ya mencionada.

2. En *inventory*, agregamos un array con los id de los *rentals* asociados. No se embebió el documento entero puesto que podría suponer una sobrecarga en la colección *inventory* en caso de que uno se rentara mucho. En base a los datos presentes, la cantidad de *rentals* por *inventory* se encuentra entre 0 y 5, con la gran mayoría entre 2 y 5. Estas cantidades justifican embeber todo el documento, pero asumiendo un caso de uso normal de la realidad modelada, consideramos oportuno no hacerlo.

3. Para las colecciones de *address*, *city* y *country*, consideramos que no precisaban su propia colección. Cada documento de las colecciones que poseen referencias a éstas, puede tener embedido los datos previamente separados. Esto disminuye considerablemente los tiempos de acceso evitando las búsquedas de documentos para estos datos pequeños y frecuentemente accedidos juntos. Este cambio introduce duplicación de datos, pero con un impacto importante en los tiempos de acceso. El contenido de las colecciones *country* y *city* pasaron a ser simples campos de tipo *string* dentro de *address*. *address* se embebe como lista en las colecciones de *store*, $staff$ y *customers*. Es importante mencionar que este cambio dificulta operaciones de modificación de datos existentes, por ejemplo si se quiere renombrar un país, pero puesto que son operaciones infrecuentes en este tipo de datos, consideramos que los beneficios son muy superiores.

4. Otra optimización que surgió de analizar los datos presentes, fue el embeber los $staff$ en la colección de *store*. Cada *store* está relacionada con una cantidad de $staff$ pequeña por lo que podemos mover el contenido de la colección $staff$ al documento de cada *store*. Este cambio también es posible porque cada $staff$ solo pertenece a una *store*.

5. Siguiendo con la colección *store*, nos encontramos con un caso particular de $staff$, el rol de manager. Luego de analizar los datos, notamos que el manager siempre es uno de los vinculados a través de $staff$. Aquí tomamos una decisión un poco particular pero que nos permitió eliminar un grado de redundancia. Cómo la lista de $staff$ fue embebida en la colección *store* y esta posee un orden fijo, se tomó la decisión de usar el índice (o posición) dentro de la lista de $staffs$ del individuo que actúa como manager. Además de no tener redundancia dentro de una misma store, nos permite tener un acceso rápido y sencillo, siempre y cuando se manipule de forma adecuada.

4 Costos con optimizaciones

En esta sección analizaremos y compararemos los resultados de las optimizaciones que realizamos respecto a los costos calculados con la base original.

4.1 Costos de customers_list

$$|customer| = n * (1 + 9 * 0.01)$$
$$|total| = 652.91$$
$$|total\_each\_day| = 65291$$

Si recordamos el costo por día original de esta consulta (736280) la mejora es bastante notable. Debido a las diferencias tan grandes en los costos en esta comparación (y en las siguientes) agregaremos una gráfica del $ActualQueryExecutionTime$ arrojado por la herramienta $explain$ de $MongoDB$. El tiempo real de las consultas nos permite visualizar claramente la ventaja de los cambios realizados. Y es que al embeber todos los datos de las otras colecciones en $customer$, reducimos los tiempo de búqueda a 0. En este caso, el $executiontime$ es tan bajo que no se aprecia. La consulta optimizada tardó $3ms$ mientras que la original concretó el mismo trabajo en $1918ms$

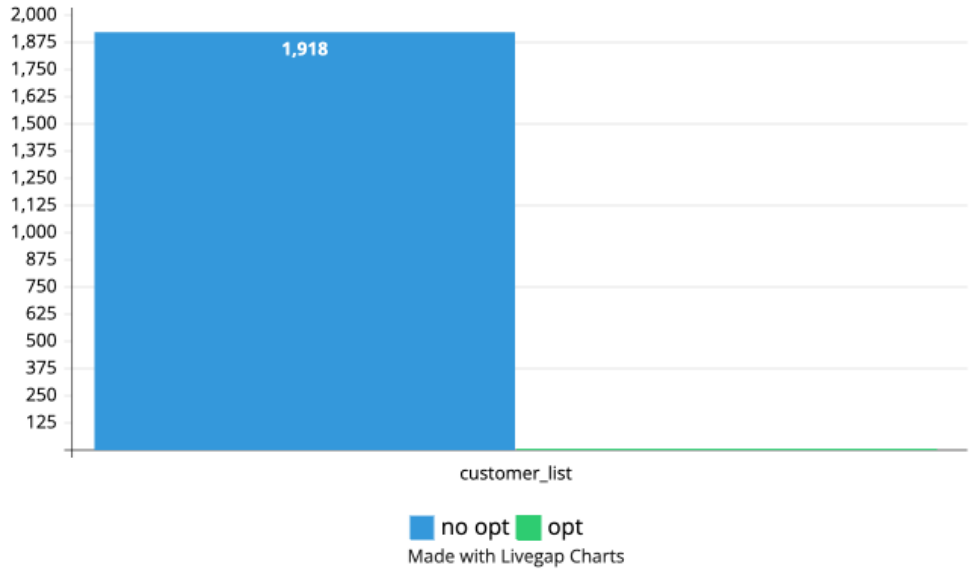

Figure 1: customer_list

4.2 Costos de film_list

Al igual que en la consulta anterior 2, aunque a menor medida, las optimizaciones de film_list mejoran sustancialmente los costos, que resultan en un tiempo menor. En esta ocasión la versión no optimizada de la consulta tardó $14380ms$ mientras que la optimizada puso $582ms$.

$$|film| = n$$
$$|actor| = n * a(1 + 0.01 + 0.01 + 0.01)$$
$$|project| = n * [(a * 0.01) + 6 * (0.01)]$$
$$|total| = n * (1 + a * 1.03 + (a * 0.01) + 0.06)|total| = 209060$$
$$|total\_each\_day| = 16724800$$

Mientras que la mejora en la anterior se debió exclusivamente a embeber los documentos, en este caso seguimos manteniendo referencias por las consideraciones ya mencionadas anteriormente. Por lo que los índices tiene un gran protagonismo en reducir los tiempos de búsqueda por referencia. En concreto el índice aprovechado fue el agregado a *actor* en su atributo *actor_id*.

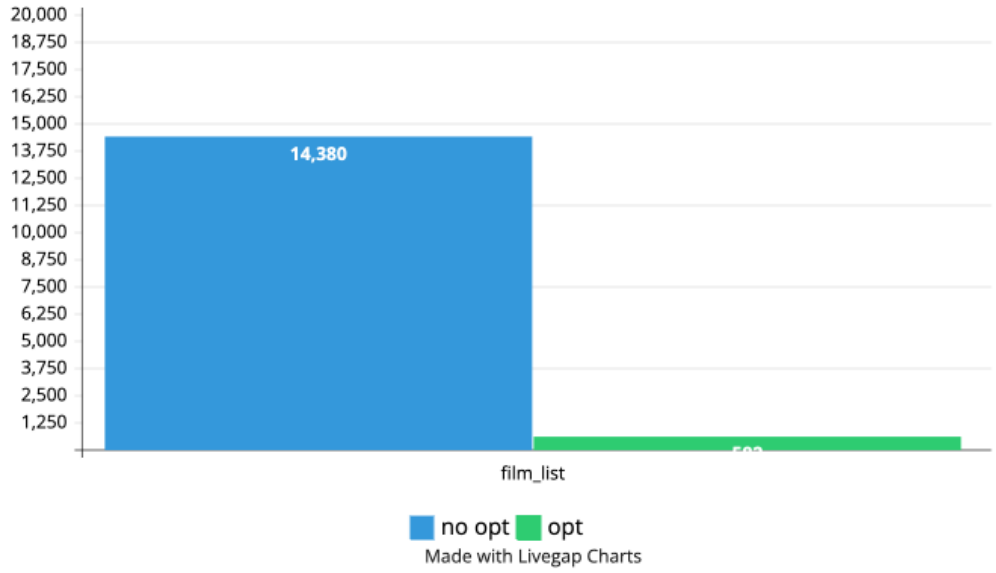

Figure 2: film_list

4.3 Costos de sales_by_store

Para esta consulta podemos observar que tenemos la misma tendencia que en las anteriores, obteniendo de nuevo mejores resultados.

$$
\begin{aligned}
|store| &= n \\
|inventory| &= n * (0.01 + i * (1 + 0.01)) \\
|rental| &= r * (0.01 + 0.01 + 1) \\
|set| &= 0.01 * 2 \\
|project| &= (9 + p) * 0.01 \\
|total| &= n + n * (0.01 + i * (1.01)) + r * (1.02) + 0.02 + (9 + p) * 0.01 \\
|total| &= 25781.07 \\
|total_{e}ach_{d}ay| &= 10312428
\end{aligned}
$$

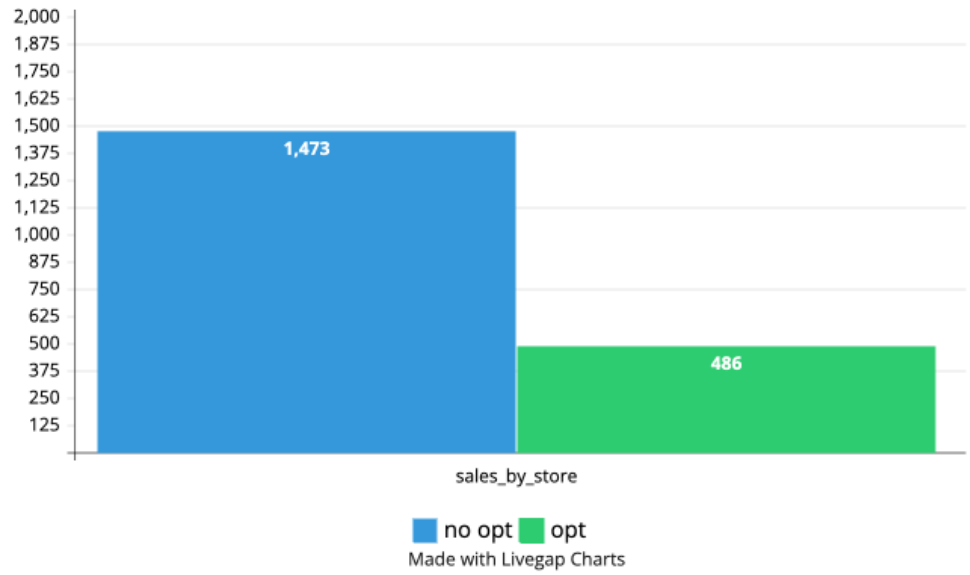

Figure 3: sales_by_store

En este caso, no se hace uso de ningún índice, debido a que las busquedas se siguen realizando sobre la clave foránea de *store*. Sin embargo, el resto de los datos embebidos de *rental*, *payment*, *address* y $staff$ permiten mejoras notables en la navegación entre las colecciones. La versión no optmizada tardó $1473ms$ y la optimizada $486ms$

4.4 Costos sales_by_film_category

Por último, volvemos a apreciar el mismo comportamiento. Los costos son nuevamente reducidos en un factor enorme, el cual impacta en los tiempos de ejecución de la misma forma.

Algo importante a notar en este estudio de tiempo, es el último project. En este, al deber utilizarse un reduce para obtener los datos necesarios de los pagos, el estudio nos lleva a tener un tiempo en esta descripto por el acceso a los *payments* y el acceso a los atributos necesarios en estos.

Notar tambien, que igual que en los casos anteriores, se utiliza el peor caso para los cálculos (la cantidad máxima de payments).

$$
\begin{aligned}
|category| &= n \\
|film| &= fc * (1 + 0.01 + 0.01) \\
|inventory| &= fc * i * (1 + 0.01) \\
|rental| &= r * (1 + 0.01) \\
|project| &= 0.01 + (p * 0.01) \\
|total| &= n + fc * 1.02 + fc * i * 1.01 + r * 1.01 + 0.01 + p * 0.01 \\
|total| &= 4644210.89 \\
|total_{e}ach_{d}ay| &= 2322105445
\end{aligned}
$$

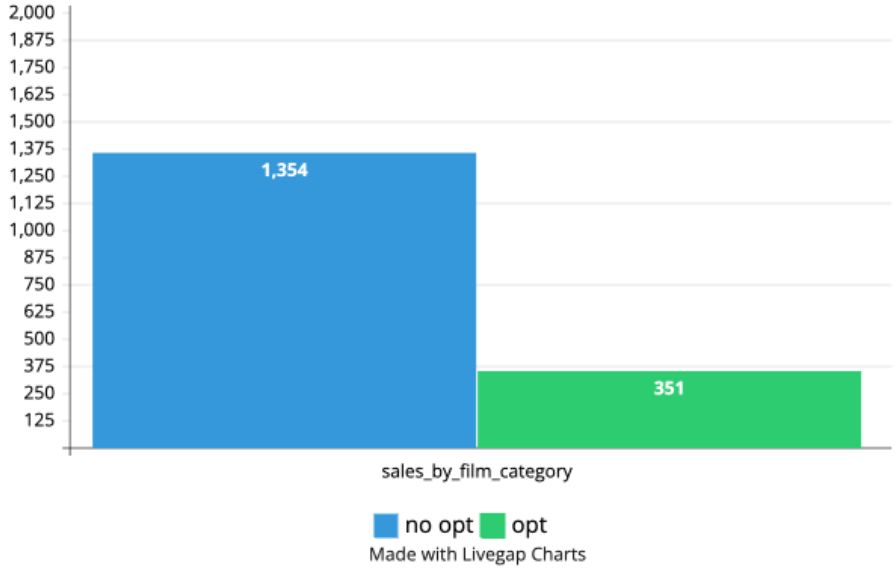

Figure 4: sales_by_film_category

En este último caso aprovechamos los índices de film aunque las mejoras son en mayor medida por embeber *payment*, *rental* y $film$. Los tiempos concretos que obtuvimos fueron: $1354ms$ para la versión no optimizada y $351ms$ para la optimizada.

5 DD1 vs DD2

Como mencionamos anteriormente, DD2, utlizando las nuevas colecciones con los respectivos vambios en las consultas, ofrece un rendimiento mejor.

Como una versión ingenua, DD1 es un buen aproximamiento para pasar de una base relacional a documental (siendo practicamente la réplica). Pero como vemos, el pasaje 1 a 1 de una base relacional no hace uso de muchas de las ventajas de las bases documentales, particularmente, $MongoDB$.

Por esto la base DD2 es preferible, y cualquier base que este diseñada con una mentalidad documental.

6 Conclusión

MongoDB y las bases documentales son una herramienta que prueban tener gran utilidad para modelar datos de realidades complejas.

Al realizar los diseños de forma beneficiosa para el motor, vemos como este tiene un gran potencial, siendo una buena opción en aplicaciones de alto volumen.

En particular, para ambientes donde se utiliza JavaScript y derivados para la implentación de aplicaciones, Mongo resulta extremadamente intuitivo y con una curva de aprendizaje relativamente baja. Si bien las filosofías y patrones que se tienen en cuenta chocan contundentemente con las enseñanzas tradicionales del mundo relacional, se puede argumentar que estas mismas razones son las que lo hacen destacar.

Sin duda es una herramienta muy poderosa y debería formar parte del abanico de cualquiera que quiera enfrentar un problema real desde el ámbito del software.