Modelado de datos del documento para bases de datos NoSQL

Mientras que las bases de datos libres de esquemas, como Azure Cosmos DB, facilitan notablemente la adopción de cambios en el modelo de datos, debe dedicar algo de tiempo a pensar en los datos.

¿Cómo se van a almacenar los datos? ¿Cómo la aplicación va a recuperar y consultar los datos? ¿La aplicación realiza muchas operaciones de lectura o escritura?

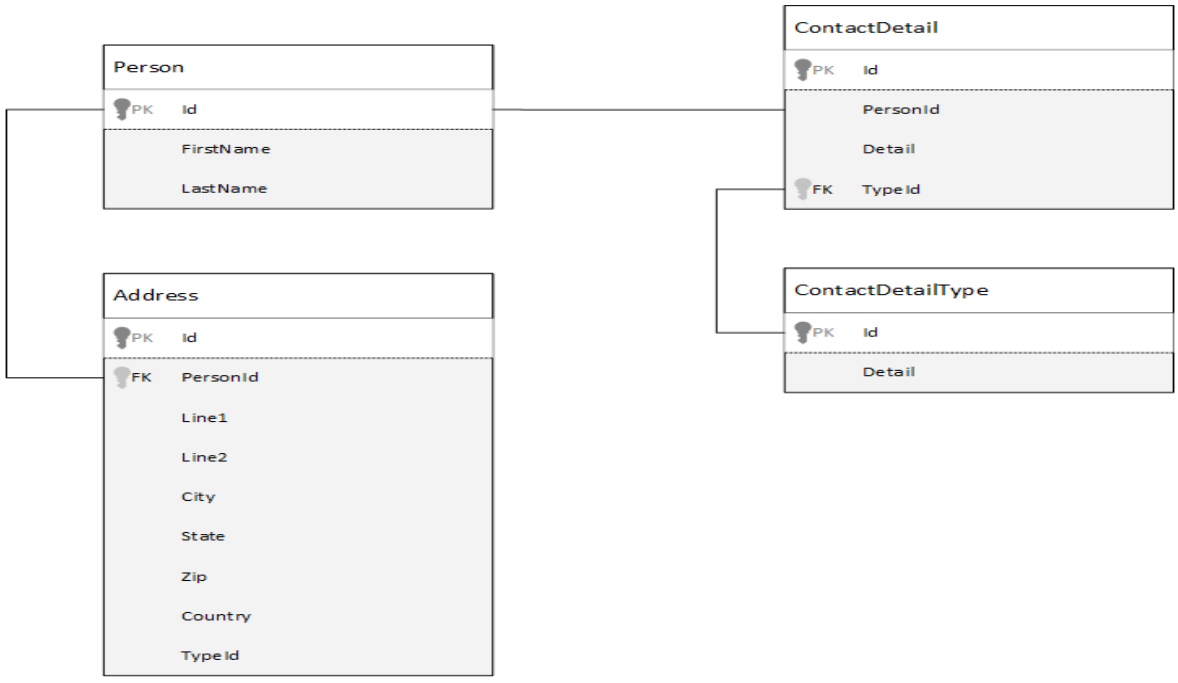
Después de leer este artículo, podrá responder a las siguientes preguntas:

- ¿Cómo debo pensar en un documento en una base de datos de documentos?
- ¿Qué es el modelado de datos y por qué tendría que importarme?
- ¿Cómo es el modelado de datos en una base de datos de documentos distinta de una base de datos relacional?
- ¿Cómo expreso relaciones de datos en una base de datos no relacional?
- ¿Cuándo se incrustan datos y cuándo realizo vinculaciones a los datos?

Incrustación de datos

Al comenzar a modelar datos en un almacén de documentos, como Azure Cosmos DB, intente tratar las entidades como **documentos independientes** representados en JSON.

Antes de adentrarnos demasiado, nos gustaría volver unos pasos atrás y echar un vistazo a cómo podríamos modelar algo en una base de datos relacional, un asunto con el que muchos de nosotros ya estamos familiarizados. En el ejemplo siguiente se muestra puede almacenarse una persona en una base de datos relacional.



Al trabajar con bases de datos relacionales, hemos aprendido durante años a normalizar constantemente.

La normalización de los datos implica normalmente tomar una entidad, como una persona, y dividirla en piezas de datos discretas. En el ejemplo anterior, una persona puede tener varios registros de información de contacto, así como varios registros de dirección. Podemos ir incluso un paso más allá y desglosar detalles de contacto extrayendo aún más campos comunes, como un tipo. Al igual que ocurre con la dirección, cada registro aquí tiene un tipo como *doméstico* o *empresarial*.

La premisa principal cuando se normalizan datos es la de **evitar el almacenamiento de datos redundantes** en cada registro y, en su lugar, hacer referencia a los datos. En este ejemplo, para leer a una persona, con toda su información de contacto y direcciones, tendrá que utilizar CONEXIONES para agregar de forma eficaz los datos en tiempo de ejecución.

```
SELECT p.FirstName, p.LastName, a.City, cd.Detail
FROM Person p
JOIN ContactDetail cd ON cd.PersonId = p.Id
JOIN ContactDetailType on cdt ON cdt.Id = cd.TypeId
JOIN Address a ON a.PersonId = p.Id
```

La actualización de una única persona con su información de contacto y direcciones requiere operaciones de escritura en muchas tablas individuales.

Ahora veamos cómo modelamos los mismos datos como una entidad independiente en una base de datos de documentos.

```
{
    "id": "1",
    "firstName": "Thomas",
    "lastName": "Andersen",
    "addresses": [
        {
            "line1": "100 Some Street",
            "line2": "Unit 1",
            "city": "Seattle",
            "state": "WA",
            "zip": 98012
        }
    ],
    "contactDetails": [
        {"email": "thomas@andersen.com"},
        {"phone": "+1 555 555-5555", "extension": 5555}
    ]
}
```

Mediante el enfoque anterior, ahora hemos **desnormalizado** el registro de la persona en el que hemos **insertado** toda la información relacionada con ella, como su información de contacto y direcciones, en un único documento JSON. Además, puesto que no estamos limitados a un esquema fijo, contamos con la flexibilidad para hacer cosas como tener información de contacto de formas diferentes por completo.

Recuperar un registro de persona completa de la base de datos es ahora una única operación de lectura frente a una colección única y para un documento único. La actualización de un registro de una persona, con su información de contacto y direcciones, es también una operación de escritura única frente a un documento único.

Mediante la desnormalización de datos, es posible que la aplicación tenga que emitir menos consultas y actualizaciones para completar operaciones frecuentes.

Cuándo se debe realizar la incrustación

Por lo general, use los modelos de datos de incrustación cuando:

- Existen relaciones contenidas** entre entidades.
- Existen relaciones de **uno a algunos** entre entidades.
- Existen datos incrustados que **cambian con poca frecuencia**.
- Existen datos incrustados que no crecerán **sin límites**.
- Existen datos incrustados que se **integran** en los datos en un documento.

Nota

Los modelos de datos desnormalizados normalmente proporcionan un mejor rendimiento de **lectura**

Cuándo no se debe incrustar

Puesto que la regla general en una base de datos de documentos es desnormalizarlo todo e incrustar todos los datos en un único documento, es posible que se produzcan situaciones que se deben evitar.

Seleccione este fragmento JSON.

```
{
    "id": "1",
    "name": "What's new in the coolest Cloud",
    "summary": "A blog post by someone real famous",
    "comments": [
        {"id": 1, "author": "anon", "comment": "something useful, I'm sure"},
        {"id": 2, "author": "bob", "comment": "wisdom from the interwebs"},
        …
        {"id": 100001, "author": "jane", "comment": "and on we go ..."},
        …
        {"id": 1000000001, "author": "angry", "comment": "blah angry blah angry"},
        …
        {"id": ∞ + 1, "author": "bored", "comment": "oh man, will this ever end?"},
    ]
}
```

Este podría el aspecto de una entidad de publicación con comentarios incrustados si se ha modelado un sistema, CMS o blog normales. El problema con este ejemplo es que la matriz de comentarios **no está limitada**, lo que significa que no hay ningún límite (práctico) para el número de comentarios que puede tener cualquier publicación única. Esto será un problema, ya que el tamaño del documento puede aumentar notablemente.

Puesto que el tamaño del documento aumenta la capacidad de transmisión de los datos a través de la conexión, así como la lectura y actualización del documento, a escala, se producirá un impacto en ellos.

En este caso sería mejor tener en cuenta el siguiente modelo.

```
Post document:
{
    "id": "1",
    "name": "What's new in the coolest Cloud",
    "summary": "A blog post by someone real famous",
    "recentComments": [
        {"id": 1, "author": "anon", "comment": "something useful, I'm sure"},
```

```
        {"id": 2, "author": "bob", "comment": "wisdom from the interwebs"},
        {"id": 3, "author": "jane", "comment": "....."}
    ]
}

Comment documents:
{
    "postId": "1"
    "comments": [
        {"id": 4, "author": "anon", "comment": "more goodness"},
        {"id": 5, "author": "bob", "comment": "tails from the field"},
        ...
        {"id": 99, "author": "angry", "comment": "blah angry blah angry"}
    ]
},
{
    "postId": "1"
    "comments": [
        {"id": 100, "author": "anon", "comment": "yet more"},
        ...
        {"id": 199, "author": "bored", "comment": "will this ever end?"}
    ]
}
```

Este modelo tiene los tres últimos comentarios insertados en la propia publicación, que es una matriz con un límite fijo esta vez. Los demás comentarios se agrupan en lotes de 100 comentarios y se almacenan en documentos independientes. El tamaño del lote se ha seleccionado como 100, puesto que nuestra aplicación ficticia permite al usuario cargar 100 comentarios a la vez.

Otro caso en el que la incrustación de datos no es una buena opción es cuando los datos incrustados se utilizan a menudo en documentos y cambian con frecuencia.

Seleccione este fragmento JSON.

```
{
    "id": "1",
    "firstName": "Thomas",
    "lastName": "Andersen",
    "holdings": [
        {
            "numberHeld": 100,
            "stock": { "symbol": "zaza", "open": 1, "high": 2, "low": 0.5 }
        },
        {
            "numberHeld": 50,
            "stock": { "symbol": "xcxc", "open": 89, "high": 93.24, "low": 88.87
}
        }
    ]
}
```

Esto podría representar la cartera de acciones de una persona. Hemos elegido insertar la información de las acciones en cada documento de la cartera. En un entorno donde los datos relacionados cambian con frecuencia, como una aplicación de comercio bursátil, la incrustación de datos que cambian con frecuencia significará que está actualizando constantemente cada documento de la cartera cada vez que se negocia una acción.

Es posible negociar con la acción *zaza* cientos de veces en un solo día y miles de usuarios pueden tener *zaza* en su cartera. Con un modelo de datos como el anterior, tendríamos que actualizar miles

de documentos de cartera varias veces al día, por lo que daría lugar a un escalado del sistema nada bueno.

Datos de referencia

Por lo tanto, la incrustación de datos funciona bien en muchos casos, pero está claro que hay escenarios en los que la desnormalización de los datos provocará más problemas que ventajas. ¿Qué hacemos ahora?

Las bases de datos relacionales no son el único lugar donde puede crear relaciones entre entidades. En una base de datos de documentos, puede tener información en un documento que realmente se relacione con los datos en otros documentos. No soy partidario ahora de dedicar ni un solo minuto a crear sistemas que podrían ser más adecuados para una base de datos relacional en Azure Cosmos DB o cualquier otra base de datos de documentos, ya que las relaciones simples son correctas y pueden ser útiles.

En el siguiente JSON hemos optado por utilizar el ejemplo de una cartera de acciones anteriores, pero esta vez hacemos referencia al artículo comercial en la cartera en lugar de incrustarlo. De esa forma, cuando el artículo comercial cambia frecuentemente a lo largo del día, el único documento que tiene que actualizarse es el documento de acciones único.

```
Person document:
{
    "id": "1",
    "firstName": "Thomas",
    "lastName": "Andersen",
    "holdings": [
        { "numberHeld":  100, "stockId": 1},
        { "numberHeld":  50, "stockId": 2}
    ]
}

Stock documents:
{
    "id": "1",
    "symbol": "zaza",
    "open": 1,
    "high": 2,
    "low": 0.5,
    "vol": 11970000,
    "mkt-cap": 42000000,
    "pe": 5.89
},
{
    "id": "2",
    "symbol": "xcxc",
    "open": 89,
    "high": 93.24,
    "low": 88.87,
    "vol": 2970200,
    "mkt-cap": 1005000,
    "pe": 75.82
}
```

Sin embargo, una desventaja de este enfoque inmediato es si la aplicación es necesaria para mostrar información sobre cada acción que se mantiene al mostrar la cartera de una persona; en este caso necesitaría realizar varios viajes a la base de datos para cargar la información de cada documento de

acciones. Aquí hemos tomado una decisión de mejorar la eficacia de las operaciones de escritura, que se producen con frecuencia a lo largo del día, pero a su vez, se comprometen las operaciones de lectura que potencialmente tienen un menor impacto en el rendimiento de este sistema en particular.

Nota

Los modelos de datos normalizados **pueden requerir varios viajes de ida y vuelta** al servidor.

¿Qué sucede con las claves externas?

Puesto que actualmente no hay ningún concepto de restricción, clave externa o de otra índole, las relaciones entre documentos que tienen en los documentos son efectivamente "puntos débiles" y la base de datos no los comprobará. Si desea asegurarse de que los datos a los que hace referencia un documento existen realmente, debe hacerlo en la aplicación o mediante el uso de desencadenadores de servidor o procedimientos almacenados en Azure Cosmos DB.

Cuándo se debe establecer referencias

Por lo general, se deben utilizar modelos de datos normalizados cuando:

- Se realiza una representación de relaciones de **uno a varios** .
- Se realiza una representación de las relaciones de **muchos a muchos** .
- Los datos relacionados **cambian con frecuencia**.
- Puede **cancelarse el límite**de los datos de referencia.

Nota

Normalmente, la normalización proporciona un mejor rendimiento de **escritura** .

¿Dónde coloco la relación?

El crecimiento de la relación le ayudará a determinar en qué documento almacenar la referencia.

Si observamos el JSON siguiente que sirve como modelo para los publicadores y los libros.

```
Publisher document:
{
    "id": "mspress",
    "name": "Microsoft Press",
    "books": [ 1, 2, 3, ..., 100, ..., 1000]
}

Book documents:
{"id": "1", "name": "Azure Cosmos DB 101" }
{"id": "2", "name": "Azure Cosmos DB for RDBMS Users" }
{"id": "3", "name": "Taking over the world one JSON doc at a time" }
...
{"id": "100", "name": "Learn about Azure Cosmos DB" }
...
{"id": "1000", "name": "Deep Dive into Azure Cosmos DB" }
```

Si el número de libros por publicador es reducido con un crecimiento limitado, almacenar la referencia del libro dentro del documento del publicador puede ser útil. Sin embargo, si el número de libros por publicador no tiene un límite, este modelo de datos llevaría provocaría matrices crecientes y mutables como ocurre en el documento de publicador de ejemplo anterior.

Cambiar las cosas un poco provocaría la creación de un modelo que seguiría representando los mismos datos, pero que evitaría esas grandes colecciones mutables.

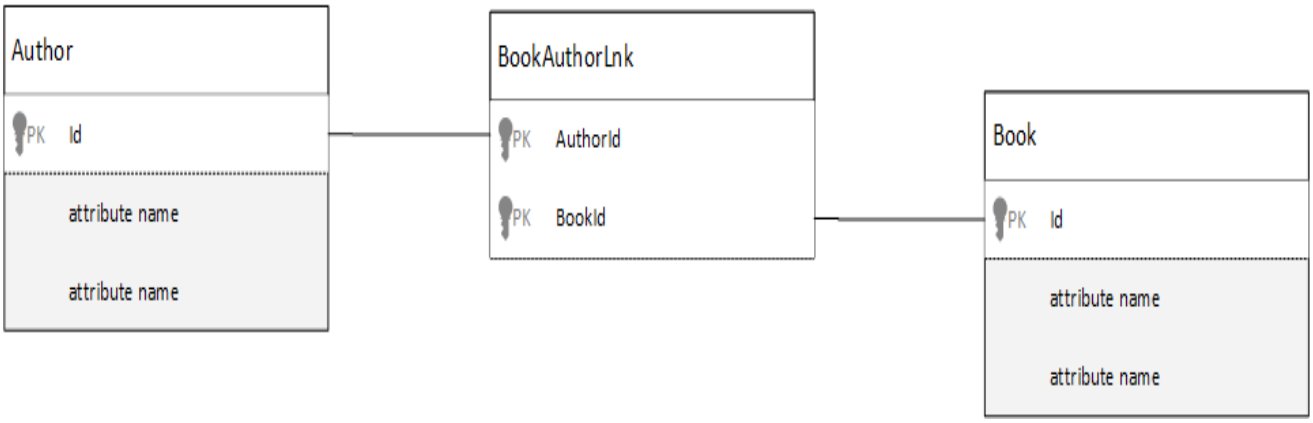
```
Publisher document:
{
    "id": "mspress",
    "name": "Microsoft Press"
}

Book documents:
{"id": "1","name": "Azure Cosmos DB 101", "pub-id": "mspress"}
{"id": "2","name": "Azure Cosmos DB for RDBMS Users", "pub-id": "mspress"}
{"id": "3","name": "Taking over the world one JSON doc at a time"}
...
{"id": "100","name": "Learn about Azure Cosmos DB", "pub-id": "mspress"}
...
{"id": "1000","name": "Deep Dive into Azure Cosmos DB", "pub-id": "mspress"}
```

En el ejemplo anterior, hemos eliminado la colección ilimitada en el documento del publicador. En su lugar, solo tenemos una referencia al publicador en cada documento del libro.

¿Cómo se puede modelar las relaciones de varios a varios?

En una base de datos relacional, las relaciones *varios a varios* modelan con frecuencia con tablas de unión, que simplemente unen registros de otras tablas.



Podría verse tentado a replicar lo mismo con documentos y producir un modelo de datos que tenga un aspecto similar al siguiente.

```
Author documents:
{"id": "a1", "name": "Thomas Andersen" }
{"id": "a2", "name": "William Wakefield" }

Book documents:
{"id": "b1", "name": "Azure Cosmos DB 101" }
{"id": "b2", "name": "Azure Cosmos DB for RDBMS Users" }
{"id": "b3", "name": "Taking over the world one JSON doc at a time" }
{"id": "b4", "name": "Learn about Azure Cosmos DB" }
{"id": "b5", "name": "Deep Dive into Azure Cosmos DB" }

Joining documents:
{"authorId": "a1", "bookId": "b1" }
{"authorId": "a2", "bookId": "b1" }
{"authorId": "a1", "bookId": "b2" }
{"authorId": "a1", "bookId": "b3" }
```

Esto funcionaría. Sin embargo, cargar un autor con sus libros o cargar un libro con su autor siempre requeriría al menos dos consultas adicionales en la base de datos. Una consulta para el documento de unión y, a continuación, otra consulta para capturar el documento real que se está uniendo.

Si todo lo que hace esta tabla de unión es combinar dos datos, ¿por qué no quitarla completamente? Tenga en cuenta lo siguiente.

```
Author documents:
{"id": "a1", "name": "Thomas Andersen", "books": ["b1, "b2", "b3"]}
{"id": "a2", "name": "William Wakefield", "books": ["b1", "b4"]}

Book documents:
{"id": "b1", "name": "Azure Cosmos DB 101", "authors": ["a1", "a2"]}
{"id": "b2", "name": "Azure Cosmos DB for RDBMS Users", "authors": ["a1"]}
{"id": "b3", "name": "Learn about Azure Cosmos DB", "authors": ["a1"]}
{"id": "b4", "name": "Deep Dive into Azure Cosmos DB", "authors": ["a2"]}
```

Ahora, si tuviera un autor, sabría de inmediato qué libros ha escrito y, a la inversa, si tuviera un documento del libro cargado sabría los identificadores de los autores. Esto ahorra la consulta intermedia de la tabla de unión, por lo que se reduce el número de viajes de ida y vuelta al servidor que tiene que realizar la aplicación.

Modelos de datos híbridos

Ya hemos observado la incrustación (o desnormalización) y el establecimiento de referencias (o normalización). Cada opción tiene sus ventajas y compromisos, como hemos visto.

No tiene por qué ser siempre una u otra. No tenga miedo de mezclarlas un poco.

En función de las cargas de trabajo y los diseños de uso específico de la aplicación, es posible que existan casos en los que mezclar los datos de referencia e incrustados tenga sentido y puede generar una lógica de aplicación más simple con menos viajes de ida y vuelta de servidor a la vez que se sigue manteniendo un buen nivel de rendimiento.

Considere el siguiente JSON.

```
Author documents:
{
    "id": "a1",
    "firstName": "Thomas",
    "lastName": "Andersen",
    "countOfBooks": 3,
     "books": ["b1", "b2", "b3"],
    "images": [
        {"thumbnail": "https://....png"}
        {"profile": "https://....png"}
        {"large": "https://....png"}
    ]
},
{
    "id": "a2",
    "firstName": "William",
    "lastName": "Wakefield",
    "countOfBooks": 1,
    "books": ["b1"],
    "images": [
        {"thumbnail": "https://....png"}
    ]
```

```
}

Book documents:
{
    "id": "b1",
    "name": "Azure Cosmos DB 101",
    "authors": [
        {"id": "a1", "name": "Thomas Andersen", "thumbnailUrl":
"https://....png"},
        {"id": "a2", "name": "William Wakefield", "thumbnailUrl":
"https://....png"}
    ]
},
{
    "id": "b2",
    "name": "Azure Cosmos DB for RDBMS Users",
    "authors": [
        {"id": "a1", "name": "Thomas Andersen", "thumbnailUrl":
"https://....png"},
    ]
}
```

Aquí hemos seguido principalmente el modelo de incrustación, donde los datos de otras entidades se incrustan en el documento de nivel superior, pero se establece la referencia en otros datos.

Si observa el documento del libro, podemos ver algunos campos interesantes cuando nos centramos en la matriz de autores. Hay un campo *id* que es el campo que se usa para volver a hacer referencia a un documento de autor, una práctica estándar en un modelo normalizado, pero también tenemos *name* y *thumbnailUrl*. Podríamos habernos parado en *id* y dejar la aplicación para obtener información adicional que necesita a partir del documento de autor correspondiente mediante el "vínculo", pero como nuestra aplicación muestra el nombre del autor y una imagen en miniatura con cada libro mostrado, podemos ahorrar un viaje de ida y vuelta al servidor por libro en una lista mediante la desnormalización de **algunos** datos del autor.

Por supuesto, si cambia el nombre del autor o se desea actualizar la foto, tendríamos que realizar una actualización de cada libro publicado, pero para nuestra aplicación, que se basa en la suposición de que los autores no cambian sus nombres a menudo, se trata de una decisión de diseño aceptable.

En el ejemplo hay valores de **agregados calculados previamente** para ahorrar un procesamiento costoso en una operación de lectura. En el ejemplo, algunos de los datos incrustados en el documento del autor son datos que se calculan en tiempo de ejecución. Cada vez que se publica un nuevo libro, se crea un documento de libro **y** el campo countOfBooks se establece en un valor calculado en función del número de documentos de libro que existen para un autor concreto. Esta optimización sería adecuada en sistemas en los que se realizan muchas operaciones de lectura, donde podemos permitirnos hacer cálculos en las escrituras para optimizarlas.

La capacidad de tener un modelo con campos calculados previamente es posible porque Azure Cosmos DB admite **transacciones de varios documentos**. Muchos almacenes NoSQL no pueden realizar transacciones en documentos y, por lo tanto, recomiendan las decisiones de diseño, por ejemplo, "siempre incrustar todo", debido a esa limitación. Con Azure Cosmos DB, puede utilizar los desencadenadores del servidor o los procedimientos almacenados que insertan los libros y actualizan los autores dentro de una transacción ACID. Ahora no **tiene** que insertar todo en un documento para asegurarse de que los datos sigan siendo coherentes.