# Quản lý mối quan hệ giữa các document

* Với RavenDB, một trong những nguyên tắc khi thiết kế database là làm cho các documents độc lập nhau, có nghĩa là tất cả thông tin được yêu cầu khi xử lý một document được lưu trữ toàn bộ trong document đó. Tuy nhiên điều đó không có nghĩa là không có các mối quan hệ (relations) giữa các đối tượng.
* Có những trường hợp mà chúng ta cần phải xác định mối quan hệ của các đối tượng. Làm như thế thì chúng ta sẽ gặp một vấn đề lớn: bất cứ khi nào chúng ta nạp dữ liệu đối tượng chứa (container object), chúng ta cần phải nạp dữ liệu cho những thực thể tham chiếu (referenced entitys) (trừ khi chúng ta không quan tâm tới nó). So với việc lưu toàn bộ thông tin cần thiết vào một thực thể thì việc tham chiếu đến thực thể có vẻ như đỡ tốn chi phí lúc đầu, nhưng điều này được chứng minh là khá tốn kém về tài nguyên dữ liệu và lưu lượng truy cập mạng.
* RavenDB cung cấp 3 phương pháp để giải quyết vấn đề này. Mỗi trường hợp sẽ sử dụng một hoặc nhiều phương pháp. Khi áp dụng một cách chính xác, các phương pháp này sẽ làm cải thiện hiệu suất, giảm băng thông và tăng tốc phát triển một cách đáng kể.

# Phương pháp tiếp cận

## Phương pháp 1: Denormalization (Chuẩn hóa ngược)

* Cách dễ nhất để giải quyết vấn đề này là chuẩn hóa ngược dữ liệu (denormalization data) vào trong thực thể chứ. Thực thể chứa sẽ chứa những dữ liệu thay cho dữ liệu tham chiếu tới.
* Xem một ví dụ về một JSON document:

|  |
| --- |
| { // Order document with id: orders/1234    "Customer": {      "Name": "Itamar",      "Id": "customers/2345"    },    Items: [      {        "Product": {          "Id": "products/1234",          "Name": "Milk",          "Cost": 2.3          },        "Quantity": 3      }    ]  } |

* Order document đã chứa dữ liệu đã được chuẩn hóa ngược của 2 documents là Customer và Product, Customer và Product ducoments được lưu trữ đầy đủ ở một nơi khác. Lưu ý là chúng ta sẽ không copy toàn bộ thuộc tính của Customer vào trong Order, chúng ta chỉ copy những thuộc tính của Customer mà chúng ta quan tâm khi cần hiển thị hay xử lý với các Order. Cách tiếp cận này được gọi là “**denormalized reference”.**
* Cách tiếp cận denormalization này giúp chúng ta tránh việc tìm kiếm chéo dữ liệu và chỉ những kết quả cần thiết mới được truyền tải qua mạng, nhưng nó lại làm cho một số trường hợp khác trở nên khó khăn. Ví dụ, lúc đầu chúng ta có những thực thể có cấu trúc như bên dưới:

|  |
| --- |
| public class Order  {      public string CustomerId { get; set; }      public string[] SupplierIds { get; set; }      public Referral Refferal { get; set; }      public LineItem[] LineItems { get; set; }      public double TotalPrice { get; set; }  }    public class Customer  {      public string Name { get; set; }      public string Address { get; set; }      public short Age { get; set; }      public string HashedPassword { get; set; }  } |

* Nếu chúng ta biết rằng khi nào chúng nạp dữ liệu cho Order từ cơ sở dữ liệu, chúng ta cũng cần nạp dữ liệu cho Customer Name và Customer Address, chúng ta có thể tạo ra một trường dữ liệu chuẩn hóa ngược Order.Customer, và lưu thông tin này trực tiếp vào trong đối tượng Order. Customer Password và những thông tin không cần thiết khác sẽ không được chuẩn hóa ngược:

|  |
| --- |
| public class DenormalizedCustomer  {      public int Id { get; set; }      public string Name { get; set; }      public string Address { get; set; }  } |

* Đây không phải là tham chiếu trực tiếp giữa Order và Customer. Order sẽ chứa một DenormalizedCustomer(sẽ chứa những thông tin cần thiết từ Customer khi chúng ta xử lý đối tượng Order).
* Nhưng chuyện gì sẽ xảy ra nếu như Customer’s Address thay đổi? Chúng ta phải thực hiện một loạt các thao tác để cập nhật lại tất cả các Order mà Customer này đã thực hiện. Và điều gì sẽ xảy ra nếu Customer này có rất nhiều Orders hoặc là địa chỉ của họ thường xuyên thay đổi? Giữ cho các thông tin được đồng bộ là yêu cầu trên máy chủ. Điều gì sẽ xảy ra nếu một quá trình làm việc khác cần một tập các thuộc tính khác của Customer ? DenormalizedCustomer cần phải được mở rộng, và như thế thì một số lượng lớn Customer record được nhân bản.

## Phương pháp 2: Includes

* Tính năng “Include” trong RavenDB nhắm đến sự hạn chếc của phương pháp “Drnormalization”. Thay vì đối tượng chứa copy những thông tin từ những đối tượng khác, nó chỉ cần giữ tham chiếu đến đôi tượng cần quan tâm. Khi đó RavenDB có thể tải dữ liệu(pre-load) cho đối tượng

được tham chiếu đến vào thời điểm mà đối tượng chính được nạp dữ liệu. Làm điều này như sau:

|  |
| --- |
| var order = session.Include<Order>(x => x.CustomerId)      .Load("orders/1234");    // this will not require querying the server!  var cust = session.Load<Customer>(order.CustomerId); |

* Với đoạn code trên, chúng ta yêu cầu RavenDB lấy dữ liệu đối tượng Order “order/1234” và cùng lúc đó thì nó sẽ “include” Customer vào Order được tham chiếu bởi thuộc tính Order.CustomerId. Phương thức Load() thứ 2 sẽ được giải quyết hoàn toàn ở phía client(không cần một yêu cầu thứ 2 tới RavenDB) bởi vì đối tượng Customer thích hợp đã được nạp dữ liệu (đây là một đối tượng Customer đầy đủ, không phải là một DenormalizedCustomer). Chúng ta cũng có thể sử dụng “Includes” với truy vấn sau:

|  |
| --- |
| var orders = session.Query<Order>()      .Customize(x => x.Include<Order>(o => o.CustomerId))      .Where(x => x.TotalPrice > 100)      .ToList();    foreach (var order in orders)  {      // this will not require querying the server!      var cust = session.Load<Customer>(order.CustomerId);  } |

* RavenDB có 2 kênh xuyên suốt mà nó trả về thông tin cho một yêu cầu nạp dữ liệu. Thứ nhất là kênh Results, đối tượng chính sẽ được nạp dữ liệu bởi phương thức Load(). Thứ 2 là kênh Includes, những documents được include sẽ được gửi về phía client. ở phía Client, những documents được include không được trả về bởi phương thức Load(), bởi vì nó đã được thêm vào session unit of work, và những yêu cầu tiếp theo để nạp dữ liệu sẽ được làm bởi session cache, không có bất cứ yêu cầu truy vấn nào gửi đến server nữa.

### One to many includes

* Include có thể được sử dụng với mối quan hệ nhiều-một (many to one). Với những lớp ở trên, một Order có thuộc tính SupplierIds là một mảng tham chiếu đến Supplier documents. Đoạn code dưới đây sẽ pre-load suppliers:

|  |
| --- |
| var order = session.Include<Order>(x => x.SupplierIds)      .Load("orders/1234");    foreach (var supplierId in order.SupplierIds)  {      // this will not require querying the server!      var supp = session.Load<Supplier>(supplierId);  } |

* Một lần nữa chúng ta lại thấy rằng, phương thức Load trong vòng lặp foreach sẽ không gửi yêu cầu đến server bởi vì các suppliers đã được load vào trong session cache.

### Secondary level includes

* Includes không chỉ làm việc với các thuộc tính ở mức 1 mà còn làm việc tốt với các thuộc tính ở mức 2. Lớp Order chứa thuộc tính Referral có cấu trúc sau:

|  |
| --- |
| public class Referral  {      public string CustomerId { get; set; }      public double CommissionPercentage { get; set; }  } |

* Lớp này có chứa CustomerId. Đoạn code dưới đây sẽ include document được tham chiếu ở mức 2:

|  |
| --- |
| var order = session.Include<Order>(x => x.Refferal.CustomerId)      .Load("orders/1234");    // this will not require querying the server!  var referrer = session.Load<Customer>(order.Refferal.CustomerId); |

* Secondary level include cũng làm việc với collections. Thuộc tính Order.LineItems giữ một tập hợp đối tượng LineItem chứa tham chiếu đến Product:

|  |
| --- |
| public class LineItem  {      public string ProductId { get; set; }      public string Name { get; set; }      public int Quantity { get; set; }      public double Price { get; set; }  } |

* Documents Product có thẻ được include bằng cách sử dụng cú pháp:

|  |
| --- |
| var order = session.Include<Order>(x => x.LineItems.Select(li => li.ProductId))      .Load("orders/1234");    foreach (var lineItem in order.LineItems)  {      // this will not require querying the server!      var product = session.Load<Product>(lineItem.ProductId);  } |

* Phương thức Select() trong Include chỉ cho RavenDB biết rằng thuộc tính ở mức 2 được sử dụng làm tham chiếu.

### ValueType identifiers (xác định kiểu dữ liệu)

* Ví dụ về Include ở trên giả dụ rằng thuộc tính Id được sử dụng để làm tham chiếu là một chuỗi và nó chứa chuỗi đầy đủ id để tham chiếu đến các documents( ví dụ như thuộc tính CustomerId sẽ chứa giá trị “customers/5678”). Include cũng làm việc với Value Type identifiers. Cùng xem những thực thể dưới đây:

|  |
| --- |
| public class Order2  {      public int Customer2Id { get; set; }      public Guid[] Supplier2Ids { get; set; }      public Referral2 Refferal2 { get; set; }      public LineItem2[] LineItem2s { get; set; }      public double TotalPrice { get; set; }  }    public class Customer2  {      public int Id { get; set; }      public string Name { get; set; }      public string Address { get; set; }      public short Age { get; set; }      public string HashedPassword { get; set; }  }    public class Referral2  {      public int Customer2Id { get; set; }      public double CommissionPercentage { get; set; }  }    public class LineItem2  {      public Guid Product2Id { get; set; }      public string Name { get; set; }      public int Quantity { get; set; }      public double Price { get; set; }  } |

* Ví dụ trên lúc nãy có thể được viết lại như sau:

|  |
| --- |
| var order = session.Include<Order2, Customer2>(x => x.Customer2Id)      .Load("orders/1234");    // this will not require querying the server!  var cust2 = session.Load<Customer2>(order.Customer2Id); |

|  |
| --- |
| var orders = session.Query<Order2>()      .Customize(x => x.Include<Order2, Customer2>(o => o.Customer2Id))      .Where(x => x.TotalPrice > 100)      .ToList();    foreach (var order in orders)  {      // this will not require querying the server!      var cust2 = session.Load<Customer2>(order.Customer2Id);  } |

|  |
| --- |
| var order = session.Include<Order2, Supplier2>(x => x.Supplier2Ids)      .Load("orders/1234");    foreach (var supplier2Id in order.Supplier2Ids)  {      // this will not require querying the server!      var supp2 = session.Load<Supplier2>(supplier2Id);  } |

|  |
| --- |
| var order = session.Include<Order2, Customer2>(x => x.Refferal2.Customer2Id)      .Load("orders/1234");    // this will not require querying the server!  var referrer2 = session.Load<Customer2>(order.Refferal2.Customer2Id); |

|  |
| --- |
| var order = session.Include<Order2, Product2>(x => x.LineItem2s.Select(li => li.Product2Id))  .Load("orders/1234");    foreach (var lineItem2 in order.LineItem2s)  {      // this will not require querying the server!      var product2 = session.Load<Product2>(lineItem2.Product2Id);  } |

* Tham số thứ 2 trong Include<T, TInclude> chỉ ra đâu là document collection được tham chiếu. RavenDB kết hợp tên của collection với thuộc tính giá trị tham chiếu để tìm ra đầy đủ của đối tượng được tham chiếu. Ví dụ như, với ví dụ đầu tiên, nếu giá trị của thuộc tính Order.Customer2Id là số nguyên 56, RavenDB sẽ include document với Id là “customer2s/56” từ database. Phương thức Session.Load<Customer2s>() sẽ nhận được giá trị 56 và sẽ tìm kiếm khi nạp dữ liệu cho document “customer2s/56” từ session cache.

## Live Projections

* Sử dụng Include thì rất hữu ích, nhưng nhiều lúc chúng ta muốn làm những thao tác phức tạp hơn. Tính năng Live Projection là duy nhất trong RavenDB, và nó có thể được coi là bước thứ 3 trong thao tác Map/Reduce: sau khi mapped(ánh xạ) tất cả dữ liệu, và nó đã được reduced (nếu index là Map/Reduce index), RavenDB server có thể chuyển đổi kết quả sang một cấu trúc hoàn toàn khác và trả về kết quả này thay vì kết quả gốc.
* Sử dụng Live Projection bạn có quyền kiểm soát nhiều hơn đối với những gì nạp vào các thực thể kết quả, và vì nó trả về projection (phép chiếu) của kết quả ban đầu, chúng ta có thể lọc ra những thuộc tính không cần thiết.
* Hãy xem một ví dụ để xem làm thế nào nó có thể được sử dụng. Giả sử chúng ta có nhiều thực thể User và nhiều trong số họ thực sự là một bí danh cho một người dùng khác. Nếu chúng ta muốn hiển thị tất cả Uers và Alias của họ sử dụng Include(), chúng ta cần phải viết như thế này:

|  |
| --- |
| // Storing a sample entity  var entity = new User {Name = "Ayende"};  session.Store(entity);  session.Store(new User {Name = "Oren", AliasId = entity.Id});  session.SaveChanges();    // ...  // ...    // Get all users, mark AliasId as a field we want to use for Including  var usersWithAliases = from user in session.Query<User>().Include(x => x.AliasId)                         where user.AliasId != null                         select user;    var results = new List<UserAndAlias>(); // Prepare our results list  foreach (var user in usersWithAliases)  {      // For each user, load its associated alias based on that user Id      results.Add(new UserAndAlias                      {                          UserName = user.Name,                          Alias = session.Load<User>(user.AliasId).Name                      }          );  } |

* Vì chúng ta sử dụng Include nên chỉ truy cập server một lần – điều đó thì tốt, nhưng toàn bộ thực thể cho những document được tham chiếu (thực thể user cho những bí danh) sẽ được trả về bởi server … và nó cần phải viết rất nhiều code.
* Using Live Projection , chúng ta có thể thu được kết quả tương tự một cách dễ dàng chỉ cần áp dụng phép biến đổi ở phía server. Đoạn code dưới đây định nghĩa một index để thực hiện Live Projection:

|  |
| --- |
| public class Users\_ByAlias : AbstractIndexCreationTask<User>  {      public Users\_ByAlias()      {          Map = users => from user in users                         select new {user.AliasId};            TransformResults =              (database, users) => from user in users                                   let alias = database.Load<User>(user.AliasId)                                   select new { Name = user.Name, Alias = alias.Name };      }  } |

* Trong hàm khai báo TransformResults sẽ được thực thi trên kết quả truy vấn, do đó chúng ta có thể chỉnh sửa, mở rộng hay lọc những kết quả này. Trong trường hợp này, Live Projection cho chúng ta thấy được dữ liệu từ những documents khác và sử dụng nó để tạo mới kiểu dữ liệu trả về.
* Live Projection trả về phép chiếu của kết quả gốc, chúng ta có thể sử dụng .As() để chuyển đổi về kiểu dữ liệu đã biết trong ứng dụng.

|  |
| --- |
| var usersWithAliases =      (from user in session.Query<User, Users\_ByAlias>()       where user.AliasId != null       select user).As<UserAndAlias>(); |

* Lợi ích chính của việc sử dụng Live Projection: chúng ta không cần phải viết nhiều code, nó được thực thi ở phía server và nó tốn ít băng thông mạng bằng cách trả về những dữ liệu mà chúng ta quan tâm.
* Lưu ý: một điểm khác biệt quan trọng là Include hữu dụng trong cả 2 trường hợp nạp dữ liệu bởi id và truy vấn dữ liệu, còn Live Projection chỉ được sử dụng cho truy vấn dữ liệu.

## Phương pháp kết hợp

* Chúng ta có thể sử dụng kết hợp các phương pháp trên. Using DenormalizedCustomer ở phần trên và tạo ra Order để sử dụng chúng:

|  |
| --- |
| public class Order3  {      public DenormalizedCustomer Customer { get; set; }      public string[] SupplierIds { get; set; }      public Referral Refferal { get; set; }      public LineItem[] LineItems { get; set; }      public double TotalPrice { get; set; }  } |

* Chúng ta đã có được những lợi ích khi sử dụng Denormalization, đơn giản và nhanh chóng load dữ liệu của Order và những thông tin cần thiết của Customer được yêu cầu khi xử lý Order. Và chúng ta cũng có thể dễ dàng và hiệu quả load đầy đủ object Customer:

|  |
| --- |
| var order = session.Include<Order3, Customer2>(x => x.Customer.Id)      .Load("orders/1234");    // this will not require querying the server!  var fullCustomer = session.Load<Customer2>(order.Customer.Id); |

* Sự kết hợp giữa Denormalization và Include có thể sử dụng với List các đối tượng Denormalized.

# Tổng kết

* Không có quy luật cụ thể nào cho việc dùng từng phương pháp trên. Nhưng ý tưởng chung là suy nghĩ theo nhiều hướng khác nhau, xem xét cái hay của từng phương pháp.
* Ví dụ như là, trong một ứng dụng thương mại điện tử, tốt hơn là chúng ta denormalized product name và product price vào trong đối tượng order line, bởi vì bạn muốn chắc chắn là Customer sẽ nhìn thấy đúng Product Name và Product Price trong lịch sử mua hàng. Nhưng Customer Name và Customer Address nên được tham chiếu thay vì Denormalized trong Order.
* Trong những trường hợp mà denormalized không phải là sự lựa chọn thì Include sẽ là phương pháp phù hợp. Bất cứ khi nào một tiến trình quan trọng được yêu cầu sau khi công việc Map/Reduce được hoàn thành hay khi chúng ta cần một cấu trúc thực thể khác được trả về hơn việc định nghĩa một index, hãy dùng Live Projections.