**ĐẠI HỌC QUỐC GIA TP. HỒ CHÍ MINH**

**TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN**

**KHOA CÔNG NGHỆ PHẦN MỀM**

**Description: C:\Users\Long Van\Desktop\uit.gif**

**KHOÁ LUẬN TỐT NGHIỆP**

**TÌM HIỂU CƠ SỞ DỮ LIỆU NOSQL VÀ**

**XÂY DỰNG ỨNG DỤNG**

Giảng viên hướng dẫn : **Ths**. **Phạm Thi Vương**

Sinh viên thực hiện : **DƯƠNG THÂN DÂN - 08520057**

**BÙI NGỌC HUY - 08520544**

Lớp : **CNPM03**

Khoá : **2008 - 2012**

***TP. Hồ Chí Minh, tháng 11 năm 2012***

**MỞ ĐẦU**

*Overview về báo cáo (tại sao lại thực hiện đề tài này?), nội dung trình bày bên dưới*

**LỜI CẢM ƠN**

Chúngemxin gửi lời cảm ơn sâu sắc đến thầy Phạm Thi Vương, đã giúp đỡ, tạo điều kiện cho nhóm hoàn thành tốt khóa luận tốt nghiệp này. Thầy đã tận tình hướng dẫn và đưa ra những nhận xét vô cùng quý giá để đề tài ngày càng hoàn thiện hơn. Những góp ý của thầy giúp cho chúng em tiếp cận, hiểu rõ và giải quyết vấn đề dễ dàng hơn.

Đồng thời, chúng em cũng xin bày tỏ lòng biết ơn đến quý thầy, cô Trường Đại Học Công Nghệ Thông Tin – Đại Học Quốc Gia Thành Phố Hồ Chí Minh, đặc biệt là các thầy, cô khoa Kỹ Thuật Phần Mềm đã tận tình truyền đạt kiến thức, kinh nghiệm cho chúng em từ những ngày đầu học tập tại trường. Sự nhiệt tình của các thầy, cô đã giúp cho chúng em có kiến thức nền tảng vững chắc cũng như kinh nghiệm thực tiễn quý báu để chúng em có thể hoàn thành tốt các nhiệm vụ học tập, làm việc và nghiên cứu.

Bên cạnh đó, chúng em cũng gửi lời cảm ơn đến gia đình, các anh, chị, bạn bè đã động viên, giúp đỡ chúng em rất nhiều trong quá trình học tập cũng như trong cuộc sống.

Thành phố Hồ Chí Minh, ngày 11, tháng 11, năm 2012

Nhóm sinh viên thực hiện

Dương Thân Dân – Bùi Ngọc Huy

**NHẬN XÉT CỦA GIẢNG VIÊN HƯỚNG DẪN**

**NHẬN XÉT CỦA GIẢNG VIÊN PHẢN BIỆN**

Mục lục

[CHƯƠNG 1: TỔNG QUAN VỀ CƠ SỞ DỮ LIỆU NOSQL 9](#_Toc340430153)

[1.1 Giới thiệu về NoSQL 9](#_Toc340430154)

[1.2 Một số khái niệm mới NoSQL 9](#_Toc340430155)

[1.3 Ưu nhược điểm của cơ sở dữ liệu NoSQL 9](#_Toc340430156)

[1.4 Cách triển khai một ứng dụng NoSQL 9](#_Toc340430157)

[CHƯƠNG 2: TÌM HIỂU CÁC GIẢI PHÁP CƠ SỞ DỮ LIỆU NOSQL 9](#_Toc340430158)

[2.1 Wide Column Store / Column Families 9](#_Toc340430159)

[2.2 Key-Value Store / Tuple Store 9](#_Toc340430160)

[2.3 Document Store 9](#_Toc340430161)

[2.4 Graph Database 9](#_Toc340430162)

[CHƯƠNG 3: TÌM HIỂU VỀ RAVENDB 10](#_Toc340430163)

[3.1 Lý do sử dụng RavenDB 10](#_Toc340430164)

[3.2 Đặc điểm của RavenDB 11](#_Toc340430165)

[3.3 Vài kĩ thuật đặc biệt khi sử dụng RavenDB 14](#_Toc340430166)

[CHƯƠNG 4: XÂY DỰNG ỨNG DỤNG SỬ DỤNG RAVENDB 14](#_Toc340430167)

[4.1 Giới thiệu về ứng dụng 14](#_Toc340430168)

[4.2 Phân tích, thiết kế hệ thống 14](#_Toc340430169)

[4.3 Triển khai 14](#_Toc340430170)

[4.4 Test performance để thể hiện sức mạnh của NoSQL 14](#_Toc340430171)

[CHƯƠNG 5: TỔNG KẾT 14](#_Toc340430172)

[5.1 Kết quả đạt được 14](#_Toc340430173)

[5.1.1 Về mặt lý thuyết 14](#_Toc340430174)

[5.1.2 Về mặt thực nghiệm 14](#_Toc340430175)

[5.2 Hướng phát triển 14](#_Toc340430176)

[Tài liệu tham khảo 14](#_Toc340430177)

**DANH MỤC CÁC BẢNG, SƠ ĐỒ, HÌNH**

# CHƯƠNG 1: TỔNG QUAN VỀ CƠ SỞ DỮ LIỆU NOSQL

## Giới thiệu về NoSQL

## Một số khái niệm mới NoSQL

## Ưu nhược điểm của cơ sở dữ liệu NoSQL

## Cách triển khai một ứng dụng NoSQL

# CHƯƠNG 2: TÌM HIỂU CÁC GIẢI PHÁP CƠ SỞ DỮ LIỆU NOSQL

Có 4 loại cơ sở dữ liệu NoSQL chính:

## Wide Column Store / Column Families

## Key-Value Store / Tuple Store

## Document Store

## Graph Database

# 

# CHƯƠNG 3: TÌM HIỂU VỀ RAVENDB



## Khái niệm cơ bản về RavenDB

* RavenDB là một công nghệ cơ sở dữ liệu dựa trên kiến trúc client-server. Dữ liệu được lưu trữ trên một thực thể máy chủ và những yêu cầu dữ liệu có thể được gửi tới máy chủ này từ một hoặc nhiều máy người dùng khác nhau.
* Những yêu cầu gửi tới máy chủ được thực hiện bằng cách sử dụng những Client API có sẵn trong bất kỳ ứng dụng .NET hoặc ứng dụng SilverLight, hoặc bằng cách truy cập trực tiếp tới Server’s RESTful API.
* Nếu là một .NET developer thì sử dụng .NET Client API là cách dễ nhất để làm việc với RavenDB vì nó cung cấp một lượng lớn các tính năng và nhiều API hỗ trợ. RESTful API làm cho RavenDB có thể được truy cập từ nhiều nền tảng khác nhau như truy vấn AJAX trong trang web hoặc là các ứng dụng Non-Windows được viết bằng Ruby-on-Rail.

## Tại sao chọn RavenDB

* RavenDB là một cơ sở dữ liệu hướng tài liệu(Document Database) mã nguồn mở có hỗ trợ transactional (giao dịch) được viết cho nền tảng .NET. RavenDB đưa ra mô hình dữ liệu linh hoạt (flexible data model) nhằm đáp ứng yêu cầu của các hệ thống thế giới thực (real-world systems). RavenDB cho phép xây dựng những ứng dụng có hiệu suất cao(high-performance), độ trễ thấp(low-latency) một cách nhanh chóng và hiệu quả.
* Dữ liệu trong RavenDB được lưu trữ dưới dạng JSON documents, phi lược đồ (scheme-less), và có thể truy vấn hiệu quả bằng cách sử dụng truy vấn Linq từ đoạn mã .NET hay sử dụng các RESTful API. RavenDB sử dụng “Index” (sẽ nói rõ hơn ở phần tiếp theo) để truy vấn dữ liệu nhanh chóng.
* RavenDB thích hợp để xây dựng các ứng dụng web-scale (các ứng dụng web có khả năng mở rộng lớn), RavenDB còn hỗ trợ replication (tạo bản sao cho các document) và sharding (phân chia dữ liệu thành các phần nhỏ lưu trên nhiều server khác nhau).

**Dưới đây là những điểm nổi bật khác của RavenDB:**

* Xây dựng ứng dụng trên cơ sở hạ tầng đã có nhằm mở rộng đáng kể kích thước của ứng dụng (RavenDB có thể lưu trữ đến 16 terrabytes trên một máy đơn).
* Chạy và làm việc tốt trên môi trường Windows. So với CouchDB thì muốn chạy CouchDB trên Windows, chúng ta cần phải biên dịch từ Erlang source code.
* RavenDB không chỉ là Server. Có thể nhúng Raven vào trong ứng dụng.
* Hỗ trợ tốt transaction. Điều này có nghĩa là RavenDB đảm bảo các tính chất ACID (Atomicity, Consistency, Isolation, Durability).
* Hỗ trợ System.Transaction và có thể thực hiện các transactions trong hệ thống phân tán.
* Cho phép định nghĩa “Indexes” sử dụng truy vấn Linq
* Hỗ trợ thực hiện thao tác map/reduce trên các documents dựa vào truy vấn Linq
* Hỗ trợ đầy đủ .NET client API, thực hiện mẫu “Unit Of Work”, thay dõi sự thay đổi, tối ưu hóa thao tác đọc/ ghi, và nhiều gói dữ liệu khác.
* Có công cụ quản lý (Raven Studio Management) giao diện web trực quan, có thể xem thao tác và truy vấn dữ liệu.
* Raven cung cấp các HTTP API để thao tác với dữ liệu trên server dựa trên giao thức REST.
* Có thể mở rộng bằng cách viết các plugins MEF(Managed Extensibility Framework).
* Hỗ trợ “partial document update” có nghĩa là không cần phải gửi toàn bộ dữ liệu của các document theo yêu cầu, chỉ gửi những dữ liệu cần thiết.
* Hỗ trợ Replication(nhân bản các document) và Sharding (phân tán dữ liệu trên nhiều server khác nhau).
* Thích hợp cho cả sản phẩm mã nguồn mở và các sản phẩm thương mại.

## Tính năng của RavenDB

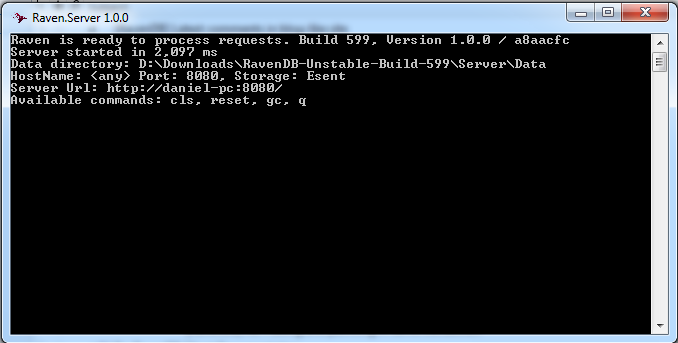
|  |  |
| --- | --- |
| Safe by default | **Safe by default**  RavenDB đảm bảo an toàn cho việc truy cập dữ liệu. Không tiêu tốn tài nguyên mạng và hệ thống. Xây dựng ứng dụng với RavenDB, tốc độ chạy chương trình nhanh và đáng tin cậy. |
| Transactional | **Transactional**  Hỗ trợ đầy đủ ACID transactions (Atomicity, Consistency, Isolation, Durability) ngay cả những node khác nhau trong hệ thống. |
| Scalable | **Scalable**  Hỗ trợ Sharding, Replication, Multi-Tenancy. Scaling out (mở rộng theo chiều ngang) tương đối dễ dàng.   * Build-in Sharding: phân tán dữ liệu trên nhiều server khác nhau để quản lý việc load dữ liệu tốt hơn. * Buil-in Replication: nhân bản dữ liệu trên nhiều server để tăng tính sẵn sàng và lấy dữ liệu nhanh chóng. * Mix replication and sharding: Có thể sử dụng kết hợp cả 2 tính năng Replication và Sharding |
| Schema free | **Schema free**  Bỏ qua những khái niệm tables, rows,mappings, complex data-layers. RavenDB là cơ sở dữ liệu hướng tài liệu, vì thế có thể lưu trữ cả đối tượng dữ liệu. |
| Get running in 5 minutes | **Get running in 5 minutes**  Chỉ cần 5 phút là đã có thể sử dụng RavenDB. RavenDB không yêu cầu cài đặt phức tạp, chỉ tải về và chạy. Rất đơn giản. |
| It Just Works | **It Just Work** |
| Fast queries | **Fast Queries**  RavenDB có thể thực hiện bất kì truy vấn với tốc độ cực nhanh(tốc độ ánh sang). Tất cả thao tác indexing được thực hiện nền (thực hiện ngầm), không ảnh hưởng đến truy vấn, thao tác đọc viết từ database. |
| Best practices built in | **Best practices built-in**   * Unit Of Work: thay đổi dữ liệu bằng cách thay đổi đối tượng nhận được từ Client API. * Domain Driven Design: mô hình dữ liệu sử dụng khái niệm DDD để thao tác dữ liệu tốt nhất. * In-memory DB for testing * Automatic-batching: tự tối ưu bằng cách gửi đi một tập lệnh thay vì một lệnh đơn. |
| High performance | **High performance**  RavenDB lưu trữ rất nhanh tất cả mô hình dữ liệu. Bỏ qua giai đoạn mapping phức tạp hay đa tầng DAL, chỉ đơn giản là lưu trữ những thực thể. |
| Caching built in | **Caching built-in**  Nhiều tầng caches thực hiện tự động trên cả server và client. Caching được cấu hình sẵn và có chế độ nâng cao là Aggressive Caching. |
| APIs | **APIs**  Có thể truy cập RavenDB bằng nhiều ngôn ngữ hay công nghệ khác nhau. Giao tiếp Client/Server thông qua REST (HTTP API), .NET client API, Silverlight and Javascript. |
| Built-in management studio | **Built-in managemet studio**  Dễ dàng quản lý dữ liệu với giao diện đồ họa trực quan. |
| Carefully designed | **Carefully designed**  RavenDB được thiết kế rất cẩn thận, tỉ mĩ đảm bảo mọi thứ hoạt động tốt. |
| Map - Reduce | **Map/Reduce**  Sử dụng indexes, dễ dàng viết các hàm Map/Reduce sử dụng cú pháp Linq. Hỗ trợ khái niệm multi-maps và boosting indexes để viết Map/Reduce đơn giản hơn và thể hiện sức mạnh của nó. |
| Feature rich and extensible | **Feature rich and extensible**  Hỗ trợ nhiều tính năng và khả năng mở rộng. |
| Embeddedable | **Embededable**  RavenDB có thể nhúng vào bất kỳ ứng dụng .NET, và nó cũng hoàn toàn phù hợp với các ứng dụng desktop. |
| Bundles | **Bundles**  Nhiều gói dữ liệu hỗ trợ đi kèm với với Server-side plugins. Chỉ cần copy file DLL vào thư mục Server. |
| Index replication to SQL | **Index replication to SQL**  Cho phép sử dụng ưu điểm của công cụ reporting có sẵn từ cơ sở dữ liệu quan hệ. RavenDB cho phép nhân bản index sang SQL table dễ dàng. |
| Full-text search built in | **Full-text Search built-in**  Không cần sử dụng công cụ hỗ trợ tìm kiếm nâng cao bên ngoài, RavenDB hỗ trợ tìm kiếm full-text ở server và client API. |
| Advanced search techniques | **Advances search techniques** |
| Geo-spatial search support | **Geo-spatial search support**  Dễ dàng sử dụng API này. |
| Easy backups | **Easy backup**  Việt lưu trữ bất đồng bộ mà không làm ảnh hưởng đến thao tác DB thông thường. Backup và Restore đều được hỗ trợ bởi DB. |
| Multi-tenancy | **Multi-tenancy**  Lưu trữ nhiều database trên một RavenDB Server. |
| Attachments | **Attachments**  RavenDB hỗ trợ lưu trữ luồng dữ liệu mà không thực sự là dữ liệu như hình ảnh hay dữ liệu nhị phân mà chúng ta không muốn lưu trữ như một document, nhưng vẫn có thể lưu trữ. |
| Online Index Rebuilds | **Online index Rebuild**  Indexes được update ngầm bên dưới mà không cần tác động của người dùng hay bất kì thao tác ACID của cơ sở dữ liệu. |
| Fully async | **Fully async (C# 5 ready)**  RavenDB hỗ trợ API bất đồng bộ mới được giới thiệu bởi C#5 |
| Community | **Community** |
| Cloud hosting available | **Cloud hosting available**  Chạy RavenDB trên đám mây với RavenHQ, CloudBird, AppHorbor hoặc Windows Azure. |

## RavenDB

### Một số thông tin cần biết về RavenDB

#### RavenDB server

* Một số cách để chạy RavenDB server:
* Chạy ứng dụng console Raven.Server.exe ( tại thư mục /Server/ trong gói sản phẩm)
* Chạy RavenDB như là một dịch vụ (service)
* Tích hợp RavenDB với IIS trên máy chủ dựa trên Windows của bạn
* Nhúng vào ứng dụng
* Để bắt đầu thì bạn cần tải gói chương trình về, giải nén, và chạy file Server/Raven.Server.exe. Bạn sẽ thấy màn hình như thế này:



* Chú ý: cổng(port) cho máy chủ để lắng nghe được chọn tự động và một thư mục dữ liệu đã được tạo và sẵn sàng để lưu trữ dữ liệu của bạn. RavenDB này chạy ở chế độ debug, để sử dụng cho quá trình tạo sản phẩm, bạn thường sẽ chạy RavenDB trong IIS hoặc chạy như một dịch vụ.
* Miễn là cửa sổ này sẽ mở, máy chủ RavenDB đã khởi động và đang chạy. Bấm Enter sẽ kết thúc máy chủ và các yêu cầu mới sẽ không được xử lý, nhưng tất cả các dữ liệu sẽ được lưu trữ trong thư mục dữ liệu.

#### Documents, Collections và Document xác định duy nhất

* Một thực thể dữ liệu duy nhất trong RavenDB được gọi là một document (tài liệu), và tất cả các tài liệu được lưu trữ trong RavenDB như các tài liệu JSON. Các định dạng JSON đã được lựa chọn vì nó có thể lưu trữ phân cấp, con người có thể đọc được. Mọi tài liệu đều có siêu dữ liệu(metadata) gắn liền với nó, theo mặc định nó chỉ chứa dữ liệu được sử dụng trong nội bộ của RavenDB (ví dụ thuộc tính Raven-Entity-Name lưu trữ các loại thực thể cho tài liệu).
* Collections là một tập hợp các tài liệu chia sẻ cùng một loại thực thể RavenDB. Nó không phải là một "bảng cơ sở dữ liệu"(database table), mà là một cách nghĩ của các nhóm tài liệu. Bộ sưu tập là một cấu trúc hoàn toàn ảo, không có ý nghĩa vật lý đối với cơ sở dữ liệu.
* Với RavenDB mỗi tài liệu có một ID riêng và duy nhất, nếu chúng ta cố gắng lưu trữ hai thực thể khác nhau theo cùng một id (ví dụ như *users/1*) – bản ghi thứ hai sẽ ghi đè lên bản ghi đầu tiên mà không có cảnh báo nào.
* Quy ước trong RavenDB: documentID được kết hợp từ tên bộ sưu tập(collection name) và id duy nhất của tài liệu trong bộ sưu tập( ví dụ users / 1). Tuy nhiên, đó chỉ là một quy ước. Document ID thì không phụ thuộc vào loại thực thể, do đó không bắt buộc phải chứa tên của bộ sưu tập chứa nó.

#### The Management Studio

* Tất cả các thực thể máy chủ có thể quản lý thông qua một ứng dụng Silverlight truy cập từ xa - Management Studio. Nó có thể được truy cập bằng cách trỏ trình duyệt của bạn đến địa chỉ và cổng máy chủ lắng nghe (mặc định là http://localhost:8080).
* Hình ảnh Management Studio:



### Hướng dẫn thêm RavenDB vào ứng dụng nhanh chóng

* Có hai cách thêm RavenDB vào ứng dụng:
  + RavenDB Client là một thư viện nhẹ cho phép bạn kết nối với bất kỳ máy chủ RavenDB, nó thể hiện sức mạnh của RavenDB ở tất cả ứng dụng .NET hoặc ứng dụng Silverlight.
  + RavenDB Embedded là một máy chủ RavenDB mạnh mẽ sẵn sàng nhúng vào trong ứng dụng của bạn.
* Nếu bạn không chắc chắn lựa chọn nào trong 2 lựa chọn trên thì hãy sử dụng RavenDB Client. Chỉ sử dụng RavenDB Embedded nếu bạn chắc chắn điều này thực sự là những gì bạn cần.
* Lưu ý: RavenDB Embedded không thể chạy trên Client Profile.

#### Thêm RavenDB sử dụng build package

* Tải về bản cài đặt (stable) từ [trang tải xuống](http://ravendb.net/download), và giải nén các tập tin cần thiết vào một thư mục mới "RavenDB" trong ứng dụng của bạn. Sau đó,trong Project bạn sẽ chọn references đến các thư viện này.
* Dưới đây các tập tin cần thiết:



* /Client – Một bản RavenDB client dành cho .NET 4.0. **Nên sử dụng bản này.**.
* /Client-3.5 - Một bản RavenDB client dành cho NET 3.5.
* /Silverlight – Một bản Silverlight 4.0 client cho RavenDB và một số phần khác đi chung.
* /EmbeddedClient – Những file cần để chạy RavenDB client, trên máy chủ hoặc là chế độ nhúng (embedded).

Dù chọn phiên bản nào để sử dụng thì nên tham chiếu đến tất cả các thư viện trong thư mục vào trong Project.

Một số thư mục còn lại:

* /Backup - [Standalone backup tool](http://ravendb.net/docs/server/administration/backup-restore), thực hiện sao lưu dữ liệu sử dụng bởi người dùng có quyền quản trị.
* /Bundles - [Bundles](http://ravendb.net/docs/server/bundles) mở rộng RavenDB theo nhiều cách khác nhau.
* /Samples – Một số ứng dụng ví dụ cho Raven.
* /Smuggler - [The Import/Export utility](http://ravendb.net/docs/server/administration/export-import) cho RavenDB.
* /Server – Những file để chạy được RavenDB trên server / service mode. Thực thi /Server/Raven.Server.exe /install để đăng kí và sử dụng dịch vụ RavenDB.
* /Web – Những file để chạy RavenDB bên dưới IIS. Tạo IIS site trong thư mục /Web để chạy RavenDB site.

#### Thêm RavenDB thông qua NuGet

##### Nuget là gì?

* NuGet là một phần mở rộng Visual Studio giúp chúng ta dễ dàng để thêm, loại bỏ, và cập nhật thư viện và các công cụ trong các dự án Visual Studio sử dụng .NET Framework. Khi bạn thêm một thư viện hoặc công cụ, NuGet sao chép các tập tin vào Solution và tự động thay đổi những thứ cần thiết trong dự án, chẳng hạn như thêm tài liệu tham khảo và thay đổi tập tin app.config hoặc web.config. Khi gỡ bỏ một thư viện, NuGet loại bỏ các tập tin và đảo ngược những thay đổi trước đó mà nó đã thực hiện trong Project.
* Tìm hiểu thêm chi tiết Nuget trên [trang web](http://docs.nuget.org/).

##### Thêm RavenDB thông qua NuGet

* Có một loạt các gói NuGet mà bạn có thể sử dụng RavenDB trong dự án của bạn. Thực hiện theo các liên kết để biết thêm thông tin về các nội dung và mục đích của mỗi gói.
* Primary Packages:
  + [RavenDB.Client](http://nuget.org/packages/RavenDB.Client)
  + [RavenDB.Embedded](http://nuget.org/packages/RavenDB.Embedded)
  + [RavenDB.AspNetHost](http://nuget.org/packages/RavenDB.AspNetHost)
* Secondary Packages:
  + [RavenDB.Database](http://nuget.org/packages/RavenDB.Database)
  + [RavenDB.Client.Debug](http://nuget.org/packages/RavenDB.Client.Debug)
  + [RavenDB.Client.FSharp](http://nuget.org/packages/RavenDB.Client.FSharp)
  + [RavenDB.Client.MvcIntegration](http://nuget.org/packages/RavenDB.Client.MvcIntegration)
  + [RavenDB.Server](http://nuget.org/packages/RavenDB.Server)
* Bundles Packages:
  + [RavenDB.Bundles.Authentication](http://nuget.org/packages/RavenDB.Bundles.Authentication)
  + [RavenDB.Bundles.Authorization](http://nuget.org/packages/RavenDB.Bundles.Authorization)
  + [RavenDB.Bundles.CascadeDelete](http://nuget.org/packages/RavenDB.Bundles.CascadeDelete)
  + [RavenDB.Bundles.Expiration](http://nuget.org/packages/RavenDB.Bundles.Expiration)
  + [RavenDB.Bundles.IndexReplication](http://nuget.org/packages/RavenDB.Bundles.IndexReplication)
  + [RavenDB.Bundles.MoreLikeThis](http://nuget.org/packages/RavenDB.Bundles.MoreLikeThis) / [Client](http://nuget.org/packages/RavenDB.Client.MoreLikeThis)
  + [RavenDB.Bundles.Replication](http://nuget.org/packages/RavenDB.Bundles.Replication)
  + [RavenDB.Bundles.Quotas](http://nuget.org/packages/RavenDB.Bundles.Quotas)
  + [RavenDB.Bundles.UniqueConstraints](http://nuget.org/packages/RavenDB.Bundles.UniqueConstraints) / [Client](http://nuget.org/packages/RavenDB.Client.UniqueConstraints)
  + [RavenDB.Bundles.Versioning](http://nuget.org/packages/RavenDB.Bundles.Versioning)
* 3rd Party:
  + Lưu ý: Những gói của hãng thứ 3 sẽ không được hỗ trợ chính thức bởi Hibernating Rhinos. Liên hệ với tác giả của những gói này nếu có những phát sinh lỗi.

### Yêu cầu hệ thống

#### Lightweight client

* Min: .NET Framework 3.5 Client Profile.
* Đề xuất: .NET Framework 4.0 Client Profile
* Client không yêu cầu Windows và **có thể chạy Mono**.

#### Silverlight client

* Silverlight 4 or 5

#### RavenDB server / Embedded

* .Net framework 4
* Windows

### Lý thuyết cơ bản RavenDB

#### Tạo khóa cho các document

RavenDB hỗ trợ các cách tạo khóa cho các document

##### RavenDB tự động tạo khóa

* Khi chúng ta không chỉ định khóa cho các document, RavenDB sẽ tự động tạo mới khóa cho các document. Raven sử dụng các GUID liên tiếp để tạo các khóa. Các GUID liên tiếp này là duy nhất và có lợi trong việc sắp xếp các indexing. Cách này thường được dùng nếu chúng ta không quan tâm tới việc tạo khóa cho các document như là lưu lại các log hay là khi mà người dùng không bao giờ hiển thị dữ liệu các khóa này.

##### Tự tạo khóa cho các document

* Chúng ta có thể gán khóa cho các document trước khi lưu các document này xuống cơ sở dữ liệu. Thường sử dụng trong các trường hợp như chúng ta muốn tạo khóa cho tập hợp người dùng trong hệ thống, ví dụ như: “user/nguyenvana”

##### Khóa xác định

* Raven xác định REST như là khóa, ví dụ “posts/1234”. Nếu bạn lưu document với khóa kết thúc bằng “/”, Raven sẽ tự động theo dấu các số xác minh cho tiền tố nếu nó không tồn tại và sẽ nối thêm các số xác minh vào khóa. Cách này được dùng hầu hết cho các trường hợp vì nó tạo ra khóa mà con người có thể đọc được.

#### Thiết kế cấu trúc document

* RavenDB lưu trữ dữ liệu không theo một lược đồ cố định, nó có lược đồ tùy ý tùy biến. Nhưng điều đó không có nghĩa rằng chúng ta không nên dành nhiều thời gian để xem xét làm thế nào để thiết kế các document để đảm bảo rằng chúng ta có thể truy cập tất cả dữ liệu chúng ta cần để phục vụ các yêu cầu của người dùng một cách hiệu quả, đáng tin cậy và chi phí bảo trì ít nhất có thể.
* Lỗi điển hình nhất mà chúng ta mắc phải là cố gắng thiết kế mô hình dữ liệu của document database giống với cách chúng ta thiết kế mô hình dữ liệu trong cơ sở dữ liệu quan hệ.
* RavenDB lưu trữ dữ liệu phi quan hệ. Cố gắng thiết kế theo mô hình quan hệ thì chúng ta sẽ có được nhiều kết quả tốt. Nhưng chúng ta sẽ đạt được kết quả vô cùng to lớn nếu sử dụng những điểm mạnh của cơ sở dữ liệu hướng document như là RavenDB.

##### Document is not flat (không như nhau)

* Documents, không giống như một dòng (row) trong RDBMS, document thì không giống nhau. Chúng ta không bị giới hạn chỉ lưu keys và values. Thay vào đó, chúng ta có thể lưu một đối tượng phức tạp như là một document. Nó có thể bao gồm arrays, dictionaries và trees. Không giống như mô hình dữ liệu quan hệ, nơi mà một dòng chỉ có thể chứa dữ liệu đơn giản và những cấu trúc dữ liệu phức tạp hơn cần được lưu trữ như là mối quan hệ, không khó để chúng ta lấy dữ liệu từ RavenDB.
* Xem ví dụ dưới đây:



* Trong cơ sở dữ liệu quan hệ, chúng ta cần không ít hơn 4 table để hiển thị dữ liệu trong một trang đơn (Posts, Comments, Tags, RelatedPosts)
* Sử dụng RavenDB, chúng ta lưu trữ tất cả thông tin chúng ta cần để làm việc như một document với cấu trúc bên dưới:



* Cấu trúc này cho phép chúng ta lấy mọi thông tin chúng ta cần để hiện thị lên trang chỉ trong một yêu cầu.

##### Raven is not relation (phi quan hệ)

* Khi bắt đầu dùng RavenDB , chúng ta sẽ gặp những vấn đề khi chúng ta cố gắng sử dụng các khái niệm của cơ sở dữ liệu quan hệ. Vấn đề chính đó là Raven phi quan hệ. Tuy nhiên, nó thực sự nhiều hơn thế, đây chỉ là một lý do tại sao RavenDB phi quan hệ.
* Raven xem mỗi document như một thực thể độc lập. Bằng cách làm như vậy, nó có thể tối ưu hóa cách các documents được lưu trữ và quản lý. Hơn nữa, một trong điểm tuyệt vời của Raven là Raven có thể lưu trữ một lượng lớn dữ liệu (quá nhiều dữ liệu để lưu trữ trên một máy tính duy nhất)
* Raven hỗ trợ sharding (phân tán dữ liệu trên nhiều server khác nhau), do đó không cần phải lưu trữ một tập hợp dữ liệu có liên quan với nhau. Mỗi document là độc lập và có thể được lưu trữ trên bất kỳ shard nào của hệ thống.
* Một khía cạnh khác về bản chất phi quan hệ của Raven là làm sao cho những documents có được đầy đủ thông tin trong chính bản thân nó. Chắc chắn là chúng ta có thể lưu trữ tham chiếu đến các document khác, nhưng nếu chúng ta cần tham khảo document khác để hiểu document hiện tại có ý nghĩa gì thì có thể là chúng ta đã sử dụng RavenDB không đúng cách.
* Với Raven, chúng ta được khuyến khích nhóm tất cả thông tin chúng ta cần vào trong một document duy nhất. Nhìn vào ví dụ trên, trong cơ sở dữ liệu quan hệ, chúng ta sẽ có một bảng liên kết cho RelatedPosts chứa id của các bài viết liên kết. Nếu muốn lấy Title của related posts, ta cần phải join vào bảng bài viết một lần nữa. Có thể làm điều đó trong Raven, nhưng đó không phải là phương pháp được đề xuất. Thay vào đó, như trong ví dụ trên, ta nên bao gồm tất cả các thông tin cần thiết vào trong document. Sử dụng phương pháp này, chúng ta có thể hiển thị trang với chỉ một yêu cầu, dẫn đến hiệu suất tổng thể tốt hơn nhiều.

##### Entities và Aggregates

* Khi nghĩ về việc Raven lưu trữ các thực thể(entities), chúng ta cần xem xét 2 quan điểm trước. Phương pháp được đề xuất là theo mẫu Aggregates từ cuốn sách [Domain Driven Design](http://domaindrivendesign.org). Một Aggregate Root chứa một vài thực thể, các loại giá trị, và kiểm soát tất cả truy cập các đối tượng được chứa trong nó. Những tham chiếu từ bên ngoài chỉ có thể tham chiếu đến Aggregate Root , không được phép tham chiếu đến các đối tượng con của nó.
* Khi áp dụng cách suy nghĩ này vào trong document database, mối tương quan giữa Aggregatre Root (trong DDD) và document trong Raven sẽ tự nhiên và dễ dàng làm theo. Một Aggregrate Root, lưu giữ tất cả đối tượng bên trong nó, sẽ là một document trong RavenDB.
* Điều này cũng giải quyết gọn gàng một vấn đề phổ biến với Aggregates: đi từ Aggregates đến các đối tượng con chứa trong nó ta cần một thao tác cụ thể, điều này tốn kém về số lượng lời gọi đến database. Trong Raven, nạp dữ liệu cho thực thể Aggragate chỉ với một lời gọi và kết hợp một document với đối tượng đầy đủ Aggregate Root thì ít tốn chi phí.
* Thay đổi sang Aggregate cũng dễ dàng hơn để kiểm soát, khi sử dụng RDBMS, thật khó khăn để đảm bảo rằng các yêu cầu đồng thời sẽ không vi phạm các business rules. Vấn đề là 2 yêu cầu riêng biệt có thể tác động đến 2 phần khác nhau của Aggregate, mỗi phần thì hợp lệ trong riêng nó, nhưng kết quả tổng hợp thì lại không hợp lệ. Điều này dẫn đến sử dụng  [coarse grained locks](http://martinfowler.com/eaaCatalog/coarseGrainedLock.html), cái mà khó thực hiện được khi sử dụng OR/Ms điển hình.
* Vì Raven xem toàn bộ Aggregate như một document duy nhất, vấn đề trên thường không tồn tại. Chúng ta có thể sử dụng hỗ trợ “optimistic concurrency” trong Raven để xác định xem Aggregate hay con của nó có thay đổi gì không. Chúng ta có thể nạp lại dữ liệu đã được thay đổi của Aggregate và thử lại các transaction.

##### Associations management

* Aggregate Root có thể chứa tất cả con của nó, nhưng Aggregates không thể tồn tại tách biệt. Ví dụ:



* Aggregate Root cho Order sẽ chứa OrderLines, nhưng một OrderLine sẽ không chứa một Product. Thay vào đó, nó chứa tham chiếu đến Product Id.
* Raven Client API sẽ không cố gắng giải quyết những mối liên kết (associations). Điều này được định trước và cũng do thiết kế. Thay vào đó, Raven giữ giá trị của khóa đối tượng được liên kết, và chỉ load các liên kết này khi thực sự cần.
* Lý do rất đơn giản: nhà thiết kế muốn một chút khó khăn khi tham chiếu dữ liệu trong những document khác. Trong OR/M chúng ta thường sử dụng: orderLine.Product.Name, nó sẽ load thực thể Product. Điều này làm chúng ta có cảm giác như khi sử dụng mô hình quan hệ, nhưng thực sự thì Raven phi quan hệ(non relation). Thiếu sót này có chủ ý từ Raven Client API nhằm nhắc nhở người dùng rằng họ nên mô hình Aggressives và Entities theo định dạng mà được để xuất để làm việc với RavenDB.

### .NET client API

#### Giới thiệu .NET client API

* Với RavenDB server, embedded hay remote, client API cho phép developer dễ dàng truy cập đến RavenDB từ bất kỳ ngôn ngữ .NET nào. Client API đưa ra tất cả khía cạnh của RavenDB tới ứng dụng của bạn một cách liền mạch.
* Để cho việc quản lý giao tiếp giữa client và server trở nên dễ dàng, client API cũng chịu trách nhiệm cho việc tích hợp đầy đủ kinh nghiệm của những người dùng các ứng dụng .NET. Trong số đó, client API chịu trách nhiệm hiện thực mẫu “Unit of work”, áp dụng quy tắc cho các tiến trình lưu trữ, nạp dữ liệu, tích hợp System.Transaction, gửi một tập yêu cầu đến server, lưu dữ liệu ở bộ nhớ đệm (caching)…
* Cách đơn giản nhất để sử dụng RavenDB là sử dụng Nuget, nhưng cũng nên tham chiếu các thư viện DLLs được cung cấp trong gói dữ liệu download từ server về.

#### Nguyên tắc thiết kế .NET client API

* RavenDB thiết kế tương đối giống với NHibrenate API. API bao gồm các lớp chính:
* IDocumentSession: Document Session dùng để thao tác với cơ sở dữ liệu, load dữ liệu từ cơ sở dữ liệu, truy vấn dữ liệu, lưu trữ và xóa dữ liệu. Đối tượng Session tạo ra tốn rất ít chi phí và là tiến trình không an toàn. Một thực thể của Interface hiện thực mẫu “Unit of Work”, theo dõi sự thay đổi cũng như nhiều tính năng khác đề cập ở trên như quản lý Transaction. Khi sử dụng .NET client API, hầu hết các thao tác với cơ sở dữ liệu đều thông qua đối tượng Session.
* IDocumentStore: Là một Session Factory, và tạo ra DocumentStore thì tốn nhiều chi phí, là tiến trình an toàn và được tạo 1 lần cho mỗi ứng dụng. Document Store chịu trách nhiệm thực sự cho các giao tiếp giữa client và server, nắm giữ các quy ước liên quan đến saving/loading dữ liệu và nhiều cấu hình cho ứng dụng, ví dụ như là http cach cho server.

#### Kết nối tới Raven data store

Như chúng ta đã biết, RavenDB có thể chạy ở 2 chế độ: chế độ client/server, giao tiếp được thực hiện thông qua HTTP và chế độ embedded (nhúng), trong đó thì client API tạo các lời gọi trực tiếp dựa vào Database API. RavenDB khuyến khích sử dụng chế độ client/server mode.

Vì một thực thể document store thì tốn chi phí để tạo ra nhưng là tiến trình an toàn nên đề xuất được đưa ra là một documentstore/ 1 database/ 1 ứng dụng.

Khi ứng dụng kết thúc, document store nên được giải phóng và xóa sạch một cách hợp lý.

##### Chạy ở server mode

* Để chạy ở chế độ server, thêm tham chiếu đến Raven.Client.Lightweight.dll trong ứng dụng, và sau khi chạy server xong chúng ta dùng đoạn code sau để kết nối tới server:

|  |  |
| --- | --- |
| |  | | --- | | var documentStore = new DocumentStore { Url = "<http://myravendb.mydomain.com/>" };  documentStore.Initialize(); | |

* "<http://myravendb.mydomain.com/>" là địa chỉ của RavenDB server

##### Chạy ở embedded mode

* Thêm tham chiếu đến Raven.Client.Embedded.dll

|  |
| --- |
| var documentStore = new EmbeddableDocumentStore  {  DataDirectory = "path/to/database/directory"  };  documentStore.Initialize(); |

##### Hỗ trợ Silverlight

* Nếu truy cập Raven từ ứng dụng Silverlight, chúng ta sẽ đến thư mục Silverlight trong gói dữ liệu và thêm tham chiếu đến tất cả các DLLs vào trong ứng dụng của chúng ta.
* Sử dụng Silverlight, chỉ có thể sử dụng RavenDB server từ bên ngoài, không sử dụng được chế độ embedded. Khởi tạo Document Store bằng cách:

|  |
| --- |
| var documentStore = new DocumentStore { Url = "<http://myravendb.mydomain.com/>" };  documentStore.Initialize(); |

##### Sử dụng chuỗi kết nối (connection string)

* Để cho mọi thứ đơn giản, Raven Client API hỗ trợ đặt tên cho chuỗi kết nối trong ứng dụng .NET.

|  |
| --- |
| new DocumentStore  {    ConnectionStringName = "MyRavenConStr"  } |

* Định nghĩa trong file app.config

|  |
| --- |
| <connectionStrings>     <add name="Local" connectionString="DataDir = ~\Data"/>     <add name="Server" connectionString="Url = [http://localhost:8080](http://localhost:8080/)"/>     <add name="Secure" connectionString="Url = [http://localhost:8080;user=beam;password=up;ResourceManagerId=d5723e19-92ad-4531-adad-8611e6e05c8a](http://localhost:8080;user=beam;password=up;ResourceManagerId=d5723e19-92ad-4531-adad-8611e6e05c8a/)"/>  </connectionStrings> |

#### Những thao tác cơ bản vơi cơ sở dữ liệu

##### Đối tượng Session

* Sau khi tạo ra Document Store, chúng ta đã sẵn sàng sử dụng cơ sở dữ liệu. Đối với bất kỳ thao tác nào tới cơ sở dữ liệu, chúng ta bắt đầu bằng cách tạo ra đối tượng Session từ Document Store. Đối tượng Session sẽ chứa tất cả mọi thứ cần để thực hiện thao tác tới cơ sở dữ liệu:

|  |
| --- |
| string companyId;  using (var session = documentStore.OpenSession())  {      var entity = new Company { Name = "Company" };      session.Store(entity);      session.SaveChanges();      companyId = entity.Id;  }    using (var session = documentStore.OpenSession())  {      var entity = session.Load<Company>(companyId);      Console.WriteLine(entity.Name);  } |

##### Mở một Session

* Khi một document store được tạo ra, bước tiếp theo là tạo ra đối tượng session dựa vào document store, điều này cho phép ta thực hiện các thao tác cơ bản CRUD. Một điều quan trọng cần lưu ý là khi gọi bất kỳ thao tác nào dựa trên session này sẽ không thực hiện thao tác dữ liệu xuống cơ sở dữ liệu cho đến khi phương thức SaveChanges() được gọi:

|  |
| --- |
| // Saving changes using the session API  using (IDocumentSession session = store.OpenSession())  {      // Operations against session        // Flush those changes      session.SaveChanges();  } |

* Trong ngữ cảnh này, chúng ta có thể nghĩ rằng session quản lý tất cả thay đổi nội tại, và SaveChanges sẽ gửi tất cả thay đổi đó tới RavenDB server. Tất cả các thao tác dữ liệu trong lời gọi SaveChanges sẽ được thực hiện(hoặc là tất cả cùng thành công, hoặc là cùng thất bại).

##### Lưu một document xuống cơ sở dữ liệu

* Trước khi bắt đâu lưu thông tin xuống RavenDB, chúng ta sẽ định nghĩa đối tượng chúng ta sẽ lưu xuống DB. Chúng ta sẽ có một số lớp bên dưới:

|  |
| --- |
| public class BlogPost  {      public string Id { get; set; }      public string Title { get; set; }      public string Category { get; set; }      public string Content { get; set; }      public DateTime PublishedAt { get; set; }      public string[] Tags { get; set; }      public BlogComment[] Comments { get; set; }  }    public class BlogComment  {      public string Title { get; set; }      public string Content { get; set; }  } |

* Để lưu một bài viết mới xuống cơ sở dữ liệu, ta sẽ tạo một mới một thực thể bài viết:

|  |
| --- |
| // Creating a new instance of the BlogPost class  BlogPost post = new BlogPost()                      {                          Title = "Hello RavenDB",                          Category = "RavenDB",                          Content = "This is a blog about RavenDB",                          Comments = new BlogComment[]                                      {                                          new BlogComment() {Title = "Unrealistic", Content = "This example is unrealistic"},                                          new BlogComment() {Title = "Nice", Content = "This example is nice"}                                        }                      }; |

* Lưu trữ bài viết vừa tạo bằng cách gọi hàm Store() và SaveChanges()

|  |
| --- |
| // Saving the new instance to RavenDB  session.Store(post);  session.SaveChanges(); |

* Hàm SaveChanges() sẽ tạo ra một giao tiếp HTTP thực sự. Lưu ý là phương thức Store() thực hiện hoàn toàn trong bộ nhớ, và chỉ có phương thức SaveChanges() mới tương tác với server:

|  |
| --- |
| POST /bulk\_docs HTTP/1.1  Accept-Encoding: deflate,gzip  Content-Type: application/json; charset=utf-8  Host: 127.0.0.1:8080  Content-Length: 378  Expect: 100-continue    [    {      "Key": "BlogPosts/1",      "Etag": null,      "Method": "PUT",      "Document": {        "Title": "Hello RavenDB",        "Category": "RavenDB",        "Content": "This is a blog about RavenDB",        "Comments": [          {            "Title": "Unrealistic",            "Content": "This example is unrealistic"          },          {            "Title": "Nice",            "Content": "This example is nice"          }        ]      },      "Metadata": {        "Raven-Entity-Name": "BlogPosts",        "Raven-Clr-Type": "BlogPost"      }    }  ]      HTTP/1.1 200 OK  Content-Type: application/json; charset=utf-8  Server: Microsoft-HTTPAPI/2.0  Date: Tue, 16 Nov 2010 20:37:00 GMT  Content-Length: 205    [    {      "Etag": "00000000-0000-0100-0000-000000000002",      "Method": "PUT",      "Key": "BlogPosts/1",      "Metadata": {        "Raven-Entity-Name": "BlogPosts",        "Raven-Clr-Type": "BlogPost",        "@id": "BlogPosts/1"      }    }  ] |

* Trong ví dụ trên ta thấy class BlogPost có thuộc tính Id và ta không gán giá trị cho nó. Đây là thuộc tính được sử dụng như là khóa chính của document. Lưu ý rằng RavenDB sẽ tạo ra khóa cho chúng ta theo quy ước mặc định, “BlogPosts/1”. Nếu document không có thuộc tính Id, RavenDB sẽ tạo ra ID duy nhất cho mỗi document và nó sẽ được lấy giá trị bằng cách gọi session.Advanced.GetDocumentId(objetc). Nói cách khác, thuộc tính Id hoàn toàn do chúng ta quyết định, vì thế ta có thể định nghĩa như một thuộc tính nếu chúng ta cần thông tin này để truy cập dữ liệu.

##### Lấy dữ liệu lên và chỉnh sửa

* Mỗi document được lưu trữ như là một phần của \_collection\_, collection là một tập hợp các document cùng loại. Vì thế, nếu đã có id của document ta có thể lấy nó lên từ database:

|  |
| --- |
| // BlogPosts/1 is entity of type BlogPost with Id of 1  BlogPost existingBlogPost = session.Load<BlogPost>("BlogPosts/1"); |

* Kết quả trong giao tiếp HTTP được thể hiện bên dưới:

|  |
| --- |
| GET /docs/BlogPosts/1 HTTP/1.1  Accept-Encoding: deflate,gzip  Content-Type: application/json; charset=utf-8  Host: 127.0.0.1:8080  HTTP/1.1 200 OK  Content-Type: application/json; charset=utf-8  Last-Modified: Tue, 16 Nov 2010 20:37:01 GMT  ETag: 00000000-0000-0100-0000-000000000002  Server: Microsoft-HTTPAPI/2.0  Raven-Entity-Name: Blogs  Raven-Clr-Type: Blog  Date: Tue, 16 Nov 2010 20:39:41 GMT  Content-Length: 214  {  "Title": "Hello RavenDB",  "Category": "RavenDB",  "Content": "This is a blog about RavenDB",  "Comments": [  {  "Title": "Unrealistic",  "Content": "This example is unrealistic"  },  {  "Title": "Nice",  "Content": "This example is nice"  }  ]  } |

* Muốn thay đổi thông tin của đối tượng ta chỉ cần làm như sau:

|  |
| --- |
| existingBlogPost.Title = "Some new title"; |

* Lưu lại những thay đổi này xuống cơ sở dữ liệu bằng cách gọi:

|  |
| --- |
| session.SaveChanges();  // chúng ta không cần gọi phương thức Update() hay theo dõi sự thay đổi của //đối tượng. RavenDB làm điều đó cho chúng ta. |

* Với ví dụ trên, kết quả lấy được trong thông báo HTTP

|  |
| --- |
| POST /bulk\_docs HTTP/1.1  Accept-Encoding: deflate,gzip  Content-Type: application/json; charset=utf-8  Host: 127.0.0.1:8080  Content-Length: 501  Expect: 100-continue  [  {  "Key": "BlogPosts/1",  "Etag": null,  "Method": "PUT",  "Document": {  "Title": "Some new title",  "Category": "RavenDB",  "Content": "This is a blog about RavenDB",  "Comments": [  {  "Title": "Unrealistic",  "Content": "This example is unrealistic"  },  {  "Title": "Nice",  "Content": "This example is nice"  }  ]  },  "Metadata": {  "Content-Encoding": "gzip",  "Raven-Entity-Name": "Blogs",  "Raven-Clr-Type": "Blog",  "Content-Type": "application/json; charset=utf-8",  "@etag": "00000000-0000-0100-0000-000000000002"  }  }  ]  HTTP/1.1 200 OK  Content-Type: application/json; charset=utf-8  Server: Microsoft-HTTPAPI/2.0  Date: Tue, 16 Nov 2010 20:39:41 GMT  Content-Length: 280  [  {  "Etag": "00000000-0000-0100-0000-000000000003",  "Method": "PUT",  "Key": "BlogPosts/1",  "Metadata": {  "Content-Encoding": "gzip",  "Raven-Entity-Name": "Blogs",  "Raven-Clr-Type": "Blog",  "Content-Type": "application/json; charset=utf-8",  "@id": "BlogPosts/1"  }  }  ] |

##### Xóa documents

* Xóa bằng cách tham chiếu đến đối tượng:
  + Khi ta lấy được document thông qua hàm load() thì chúng ta có thể xóa được document thông qua hàm delete():

|  |
| --- |
| session.Delete(existingBlogPost);  session.SaveChanges(); |

* + Kết quả trong giao tiếp HTTP được thể hiện bên dưới:

|  |
| --- |
| POST /bulk\_docs HTTP/1.1  Accept-Encoding: deflate,gzip  Content-Type: application/json; charset=utf-8  Host: 127.0.0.1:8081  Content-Length: 49  Expect: 100-continue  [  {  "Key": "blogs/1",  "Etag": null,  "Method": "DELETE"  }  ] |

* Xóa dựa vào khóa:
  + Dùng lệnh Defer của tính năng Advanced session

|  |
| --- |
| session.Advanced.Defer(new DeleteCommandData {Key = "posts/1234"}); |

* + Dùng DatabaseCommands:

|  |
| --- |
| session.Advanced.DatabaseCommands.Delete("posts/1234", null); |

##### Truy vấn cơ bản trong RavenDB

* Dữ liệu được lưu trữ trong RavenDB, chúng ta có thể lấy document lên bằng Id, cập nhật dữ liệu hay xóa đi. Một thao tác hữu dụng tiếp theo là khả năng truy vấn dựa vào cách mà document được lưu xuống cơ sở dữ liệu:
* Ví dụ như chúng ta cần truy vấn tất cả các bài viết blog theo danh mục xác định

|  |
| --- |
| var results = from blog in session.Query<BlogPost>()                where blog.Category == "RavenDB"                select blog; |

* Hoặc là với cú pháp khác, ta có thể lấy những bài viết có ít nhất 10 comments:

|  |
| --- |
| var results = session.Query<BlogPost>()      .Where(x => x.Comments.Length >= 10)      .ToList(); |

#### Truy vấn

##### Sử dụng Linq để truy vấn dữ liệu RavenDB

* Như ta đã biết ở phần trước, truy vấn được thực hiện trên collection và sử dụng đối tượng session. Nếu người dùng không chỉ định cho RavenDB biết sẽ dùng index để truy vấn dữ liệu thì RavenDB sẽ tìm index thích hợp nhất để truy vấn, hoặc ra tạo ngầm bên dưới nếu nó chưa tồn tại. Chúng ta sẽ xem indexes được tạo ra như thế nào(còn gọi là static index) trong phần tiếp theo. Không cần quan tâm index thực sự được truy vấn như thế nào, truy vấn thường được thực hiện bằng cách sử dụng LINQ
* Giả sử chúng một số thực thể được lưu trong database như sau:

|  |
| --- |
| public class Employee  {      public string Name { get; set; }      public string[] Specialities { get; set; }      public DateTime HiredAt { get; set; }      public double HourlyRate { get; set; }  }    public class Company  {      public string Id { get; set; }      public string Name { get; set; }      public List<Employee> Employees { get; set; }      public string Country { get; set; }      public int NumberOfHappyCustomers { get; set; }  } |

* Một số truy vấn cơ bản:
  + Lấy toàn bộ thực thể Company vào trong List (lưu ý là truy vấn này sẽ thực hiện .Take(128), bởi vì chúng ta không chỉ định là số thực thể cần lấy, mặc định là 128):

|  |
| --- |
| var results =      (          from company in session.Query<Company>()          select company      )          .ToArray(); |

* + Vì sử dụng Linq nên chúng ta có thể lọc dữ liệu một cách hiệu quả dựa vào biểu thức Where():

|  |
| --- |
| // lọc dữ liệu bằng cách so sánh chuỗi  var results = from company in session.Query<Company>()                where company.Name == "Hibernating Rhinos"                select company;    // lọc dữ liệu theo miền dữ liệu số  results = from company in session.Query<Company>()            where company.NumberOfHappyCustomers > 100            select company;    // lọc dữ liệu theo dữ liệu chứa trong thực thể  results = from company in session.Query<Company>()            where company.Employees.Count > 10            select company; |

* + Linq chỉ là cú pháp, bên dưới tất cả các truy vấn sẽ được chuyển thành một chuỗi các lời gọi và các biểu thức lambda. Ví dụ, đoạn code trên được viết lại giống như lúc trước khi biên dịch:

|  |
| --- |
| // lọc dữ liệu bằng cách so sánh chuỗi  var results = session.Query<Company>()      .Where(x => x.Name == "Hibernating Rhinos");    // lọc dữ liệu theo miền dữ liệu số  results = session.Query<Company>()      .Where(x => x.NumberOfHappyCustomers > 100);    // lọc dữ liệu theo dữ liệu chứa trong thực thể  results = session.Query<Company>()      .Where(x => x.Employees.Count > 10); |

* Thêm một số lựa chọn lọc dữ liệu:

|  |
| --- |
| // Trả về những company nào có ít nhất một employee tên là “Ayende” IQueryable<Company> companies = from c in session.Query<Company>()                   where c.Employees.Any(employee => employee.Name == "Ayende")                   select c;    // Truy vấn dữ liệu trên thực thể con chứa trong thực thể cần truy vấn  // Trả về những company nào có ít nhất một developer có chuyên ngành là C#  companies = from c in session.Query<Company>()              where c.Employees.Any(x => x.Specialities.Any(sp => sp == "C#"))              select c;    // sử dụng toán tử In – trả về những company nào có dữ liệu nằm trong danh sách  // được cung cấp  companies = from c in session.Query<Company>()              where c.Country.In(new [] {"Israel", "USA"})              select c; |

* Projections(phép chiếu): là chiếu một số thuộc tính được chỉ định từ document sử dụng phương thức select() của Linq, nó không phải là đối tượng ban đầu mà là một đối tượng mới được tạo ra và chứa kết quả lấy được từ câu truy vấn. RavenDB hỗ trợ projections nhưng một điều cần lưu ý là kết quả phép chiếu không được theo dõi sự thay đổi.

|  |
| --- |
| // Trong ví dụ này, chúng ta chỉ quan tâm đến thuộc tính Name của đối tượng, vì thế ta sẽ dùng phép chiếu tạo ra một đối tượng vô danh  var companyNames = from c in session.Query<Company>()                     where c.Employees.Any(x => x.Specialities.Any(sp => sp == "C#"))                     select new {c.Name}; // phép chiếu xảy ra tại đây    //giống như truy vấn trên nhưng chúng ta muốn lấy kết quả là một đối tượng Company  //chỉ thuộc tính Name có dữ liệu, những thuộc tính khác thì trống.  Company[] companies = (from c in session.Query<Company>()                         where c.Employees.Any(x => x.Specialities.Any(sp => sp == "C#"))                         select new Company {Name = c.Name}) // phép chiếu xảy ra ở đây      .ToArray(); |

* + Phép chiếu hữu dụng khi thao tác chỉ cần một phần dữ liệu. Bất cứ khi nào không cần theo dõi sự thay đổi thì phép chiếu được khuyến khích dùng để làm giảm băng thông giữa chương trình và máy chủ.
* Sắp xếp: có thể sử dụng mệnh đề orderby / .OrderBy() / .OrderByDescending() để thực hiện việc sắp xếp.
* Toán tử tập hợp (Aggregate operations): Linq hỗ trợ Count() và Distinct(). Nếu phức tạp hơn có thể dùng map/reduce indexes.

##### Phân trang

* Phân trang, là quá trình phân chia tập hợp dữ liệu thành nhiều trang hiển thị, cho phép đọc một trang tại một thời điểm. Nó rất hữu ích cho việc tối ưu băng thông, tối ưu sử dụng phần cứng hay đơn giản là người dùng không thể cùng một lúc kiểm soát được một lượng lớn dữ liệu.
* RavenDB làm cho việc phân trang trở nên dễ dàng hơn. Rất đơn giản để chỉ định kích thước trang và điểm bắt đầu. Sử dụng Linq để thực hiện việc này:

|  |
| --- |
| // Giả sử kích thước trang là 10, chúng ta sẽ lấy dữ liệu trang thứ 3 như sau:  var results = session.Query<BlogPost>()      .Skip(20) // bỏ qua 2 trang đầu tiên      .Take(10) // lấy dữ liệu 10 bài viết cho trang thứ 3      .ToArray(); // thực hiện truy vấn |

* Tính tổng số kết quả khi phân trang:
  + Khi phân trang, nhiều lúc chúng ta muốn biết chính xác số kết quả trả về của truy vấn. Client API hỗ trợ việc này:

|  |
| --- |
| RavenQueryStatistics stats;  var results = session.Query<BlogPost>()      .Statistics(out stats)      .Where(x => x.Category == "RavenDB")      .Take(10)      .ToArray();  var totalResutls = stats.TotalResults; |

* Khi truy vấn trả về 10 kết quả, totalResults sẽ giữ tổng số documents tương ứng.
* Phân trang khi có kết quả trùng lặp:
  + Đối với một số truy vấn RavenDB bỏ qua một số kết quả nội tại, và vì thế giá trị của TotalResults không còn đúng nữa. Ví dụ như thực hiện truy vấn Distinct, TotalResults sẽ chứa tổng số documents documents được tìm thấy, nhưng không đưa vào những kết quả đã bị Distinct bỏ qua.
  + Bất cứ khi nào giá trị SkippedResults lớn hơn 0 có nghĩa là chúng ta đã bỏ qua một số kết quả trong index.
  + Để phân trang đúng trong những trường hợp này, chúng ta sử dụng SkippedResults để báo cho RavenDB biết là bao nhiêu documents bị bỏ qua. Nói cách khác, với mỗi trang thì điểm bắt đầu sẽ là: .Skip(currentPage \* pageSize + SkippedResults).
  + Ví dụ sau với kích thước trang là 10:

|  |
| --- |
| RavenQueryStatistics stats;    // trang đầu tiên  var results = session.Query<BlogPost>()      .Statistics(out stats)      .Skip(0 \* 10) // lấy kết quả cho trang đầu tiên      .Take(10) // kích thước trang là 10      .Where(x => x.Category == "RavenDB")      .Distinct()      .ToArray();  var totalResutls = stats.TotalResults;  var skippedResults = stats.SkippedResults;    // trang thứ hai  results = session.Query<BlogPost>()      .Statistics(out stats)      .Skip((1 \* 10) + skippedResults) // lấy kết quả cho tranh //thứ hai và có đưa vào số lượng trang bị bỏ qua      .Take(10) // kích thước trang là 10      .Where(x => x.Category == "RavenDB")      .Distinct()      .ToArray();    // tiếp tục như thế... |

##### Stale index (index chứa kết quả cũ, chưa cập nhật)

* RavenDB thực hiện việc đánh chỉ mục dữ liệu với một tiến trình nền bên dưới chương trình, nó sẽ được thực thi bất cứ khi nào có dữ liệu mới hoặc dữ liệu cũ được chỉnh sửa, cập nhật. Tiến trình chạy nền bên dưới này cho phép server đáp ứng yêu cầu một cách nhanh chóng ngay cả khi một khối lượng lớn dữ liệu bị thay đổi. Tuy nhiên trong trường hợp này, chúng ta sẽ truy vấn với stale index.
* Khái niệm “stale index” xuất phát từ sự nhìn nhận sâu sắc về thiết kế của RavenDB, giả sử người dùng không bao giờ bị thiệt hài khi phân chia những công việc lớn trên server. Theo như RavenDB quan tâm thì việc có một kết quả cũ tốt hơn là việc mất kết nối với dữ liệu (it is better to be stale than offline). Và như vậy, nó sẽ trả về kết quả truy vấn ngay cả khi nó biết là không thể cho một kết quả truy vấn tốt nhất(up-to-date).
* Và quả thực là RavenDB trả về kết quả nhanh chóng cho bất cứ yêu cầu của người dùng, ngay cả khi liên quan đến việc đánh lại chỉ mục của hang tram hang ngàn documents. Và bởi vì yêu cầu thứ nhất sẽ được đáp ứng rất nhanh, những truy vấn tiếp theo có thể được thực hiện sau đó vài mili giây và kết quả vẫn được trả về, tuy nhiên nó được đánh dấu là Stale.
* Kiểm tra kết quả stale:
  + Là một phần của đáp ứng yêu cầu khi truy vấn index, một thuộc tính được đính kèm cho biết kết quả đó đã cũ, có nghĩa là bất kể hiện tại có một công việc nào chưa được hoàn thành bởi index đó. Có thể lấy được kết quả dựa vào đối tượng RavenQueryStatistics:

|  |
| --- |
| RavenQueryStatistics stats;  var results = session.Query<Product>()      .Statistics(out stats)      .Where(x => x.Price > 10)      .ToArray();    if (stats.IsStale)  {      // Những kết quả cũ  } |

* Khi giá trị IsStale là true thì có nghĩa là có ai đó đã thêm hoặc thay đổi Product và index không có đủ thời gian để cập nhật lại thay đổi trước khi chúng ta truy vấn.
* Trong hầu hết các trường hợp thì chúng ta không cần quan tâm đến điều đó. Những cũng có những trường hợp mà chúng ta không thể làm việc với những dữ liệu cũ.
* Lấy kết quả mới( non-stale):
  + Khi một truy vấn yêu cầu lấy kết quả non-stale, ta có thể chỉ định rõ ràng khi truy vấn:

|  |
| --- |
| RavenQueryStatistics stats;  var results = session.Query<Product>()      .Statistics(out stats)      .Where(x => x.Price > 10)      .Customize(x => x.WaitForNonStaleResults(TimeSpan.FromSeconds(5)))      .ToArray(); |

* + Lưu ý là trên ví dụ trên thì thời gian chờ là 5 giây. Ta có thể yêu cầu RavenDB chờ vô thời hạn cho đến khi nhận được kết quả non-stale, nhưng điều này chỉ nên sử dụng trong unit-testing và không bao giờ dùng trong những ứng dụng thực tế trừ khi chúng ta hiểu 100% về nó hoặc đó là điều chúng ta mong muốn.
* Thiết lập điểm ngắt khi chờ kết quả non-stale:
  + Ngay cả khi sử dụng WaitForNonStaleResults với thời gian chờ như ví dụ trên thì vẫn có khả năng nhận được kết quả cũ – ví dụ như khi thực thi một công việc đánh chỉ mục rất dài. Một cách tốt hơn để chắc chắn là bạn đang làm việc với những kết quả non-stale là sử dụng điểm ngắt và thông báo cho server biết sử dụng chúng như những thứ cơ bản:

|  |
| --- |
| RavenQueryStatistics stats;  var results = session.Query<Product>()      .Statistics(out stats)      .Where(x => x.Price > 10)      .Customize(x => x.WaitForNonStaleResultsAsOf(new DateTime(2011, 5, 1, 10, 0, 0, 0)))      .ToArray(); |

* + Điều này đảm bảo rằng chúng ta sẽ có được kết quả mới nhất cho đến thời điểm chúng ta đưa ra. Tất cả các công việc chưa hoàn thành khi có thay đổi xảy ra sẽ không được xem xét tới. Và cũng như trước là nên có một thời gian chờ.

|  |
| --- |
| WaitForNonStaleResultsAsOfNow cũng được sử dụng, nó tương đương với callingWaitForNonStaleResultsAsOf(DateTime.Now). |

* + Một lựa chọn khác là sử dụng  WaitForNonStaleResultsAsOfLastWrite. Nó sẽ theo vết thao tác viết cuối cùng của chương trình và sử dụng làm điểm ngắt. Việc này được khuyến khích sử dụng khi bạn làm việc trên máy mà đồng bộ hóa clock có thể gây ra những vấn đề, bởi vì WaitForNonStaleResultsAsOfLastWrite không sử dụng đồng hồ của máy mà nó sử dụng giá trị etag cho thao tác viết.
  + Chúng ta có thể thiết lập cho document store luôn đợi cho đến khi kết quả mới nhất như sau:

|  |
| --- |
| documentStore.Conventions.DefaultQueryingConsistency = ConsistencyOptions.MonotonicRead; |

##### Static index

* Như đã đề cập ở phần trước, RavenDB luôn sử dụng index để truy vấn dữ liệu. Bất cứ khi nào người dùng yêu cầu truy vấn dữ liệu, RavenDB sẽ sử dụng index đã có sẵn hoặc là tạo mới index nếu không có index nào phù hợp với yêu cầu.
* RavenDB cũng cho phép chúng ta tự định nghĩa index và truy vấn nó một cách tường minh. Những index người dùng tự tạo được gọi là static index.
* Một số lý do mà static index hay được sử dụng hơn những index được tạo tự động là:
  + Độ trễ thấp: Tạo index không phải là quá trình ít tốn chi phí, mà nó tốn một thời gian để thực hiện. Vì những dynamic index được tạo cùng với truy vấn đầu tiên, nên kết quả non-stale cho lần đầu truy vấn sẽ tốn nhiều thời gian trả về. Dynamic index được tạo như là những index tạm thời, điều này dẫn đến hiệu suất khi thực hiện truy vấn lần đầu.
  + Linh hoạt: Static index được hỗ trợ them nhiều chức năng khác như sorting, boosting, Full text Search, Live Projection, spatial search support …
* Trong khi sử dụng dynamic index thì sẽ dễ dàng cho chúng ta, việc sử dụng static index thì hữu dụng và hiệu quả hơn với dữ liệu thời gian thực. Vì thế, nên sử dụng static index trong hầu hết các thao tác của chương trình hay ít nhất cũng chắc chắn rằng những index tạm thời được tạo từ những dynamic index sẽ được chỉ định là những index thường dùng.
* Bất cứ khi nào chúng ta yêu cầu RavenDB truy vấn dữ liệu và đã có static index thích hợp tồn tại, RavenDB sẽ trực tiếp truy vấn sử dụng index đó một cách tự động. Chúng ta cũng có thể chỉ định tên của index mà chúng ta muốn dùng:

|  |
| --- |
| var results = session.Query<BlogPost>("MyBlogPostsIndex").ToArray(); |

* + Lưu ý là RavenDb sẽ ném ra lỗi nếu chúng ta chỉ định tên của index được sử dụng mà index này lại không thực sự tồn tại.

###### Định nghĩa static index

* Định nghĩa một static index cho phép chúng ta có những truy vấn phức tạp và nó cũng làm giảm đi những kết quả cũ trong một số trường hợp. Static index cũng thường được sử dụng hơn là dựa vào những dynamic index.
* Để định nghĩa một index, chúng ta cần một đối tượng IndexDefinition và đưa nó vào cơ sở dữ liệu. Khi nhận biết được đó là một index mới thì RavenDB server sẽ thực hiện một nhiệm vụ bên dưới nền để tạo ra index. Một index có thể được truy vấn ngay lập tức sau khi quá trình tạo index được bắt đầu, nhưng cho đến lúc quá trình này hoàn thành thì kết quả trả về sẽ được đánh dấu là stale. Index sẽ được cập nhật liên tục khi có bất kì thao tác thêm hay sửa dữ liệu nào.
* Lớp IndexDefinition:
  + Một định nghĩa index bao gồm tên index, hàm map/reduce, một hàm tùy chọn TransformResults và một vài tùy chọn khác. Cấu trúc lớp IndexDefinition được thể hiện bên dưới:

|  |
| --- |
| class IndexDefinition  {      /// <summary>      /// Get or set the name of the index      /// </summary>      public string Name { get; set; }        /// <summary>      /// Gets or sets the map function      /// </summary>      /// <value>The map.</value>      public string Map { get; set; }        /// <summary>      /// Gets or sets the reduce function      /// </summary>      /// <value>The reduce.</value>      public string Reduce { get; set; }        /// <summary>      /// Gets or sets the translator function      /// </summary>      public string TransformResults { get; set; }        /// <summary>      /// Gets or sets the stores options      /// </summary>      /// <value>The stores.</value>      public IDictionary<string, FieldStorage> Stores { get; set; }        /// <summary>      /// Gets or sets the indexing options      /// </summary>      /// <value>The indexes.</value>      public IDictionary<string, FieldIndexing> Indexes { get; set; }        /// <summary>      /// Gets or sets the sort options.      /// </summary>      /// <value>The sort options.</value>      public IDictionary<string, SortOptions> SortOptions { get; set; }        /// <summary>      /// Gets or sets the analyzers options      /// </summary>      /// <value>The analyzers.</value>      public IDictionary<string, string> Analyzers { get; set; }  } |

* + Bất kỳ index nào cũng yêu cầu phải có tên và hàm Map. Hàm Map là cách mà chúng ta thông báo cho RavenDB biết làm thế nào tìm được những dữ liệu chúng ta cần đến và những trường dữ liệu nào mà chúng ta sẽ tìm kiếm. Hàm Map được viết theo cú pháp Linq.
  + Hàm Reduce là một tùy chọn, được viết và thực thi giống như hàm Map nhưng được thực thi trên kết quả của hàm Map. Hàm Reduce thực sự là một index thứ hai cho phép chúng ta thực hiện các thao tác tập hợp ít tốn chi phí(rất rẻ) và trực tiếp từ index.
  + Hàm thứ ba là hàm TransformResults, một tính năng được gọi là Live Projections, sẽ được nói rõ ở phần sau.
  + Những thuộc tính còn lại hữu ích cho việc tận dụng toàn bộ sức mạnh của Lucene bằng cách tùy biến các indexes.
* Tạo mới một index:
  + Dùng hàm PutIndex có trong đối tượng DocumentCommands để tạo index:

|  |
| --- |
| // tạo một index mà chúng ta sẽ tìm kiếm dữ trên thuộc tính Post Title  documentStore.DatabaseCommands.PutIndex("BlogPosts/ByTitles",                                        new IndexDefinitionBuilder<BlogPost>{                                         Map = posts => from post in posts                                             select new { post.Title }                                          }); |

* + Có thể tạo một index (index class) bằng cách thừa kế từ AbstractIndexCreationTask<T> . Trong contructor của lớp này, chúng ta có thể truy cập tất cả các thuộc tính index, có thể thay đổi cho phù hợp với yêu cầu của bạn.
  + Thông báo cho server tạo ra index thực sự bằng cách thêm lời gọi vào lúc ứng dụng khởi động. Câu lệnh bên dưới sẽ gửi tất cả các lớp  AbstractIndexCreationTask<T> tới server để tạo ra các indexes (các index đã tồn tại vẫn bị ảnh hưởng):

|  |
| --- |
| IndexCreation.CreateIndexes(typeof(MyIndexClass).Assembly, documentStore); |

* + Với cách này thì mỗi index sẽ được lưu riêng ra một file, như thế sẽ dàng cho chúng ta làm việc với index trong trường hợp có quá nhiều index.
* Thêm index trên thực tế:
  + Giả sử chúng ta có một cái blog với nhiều bài viết, mỗi bài viết có một vài tags, và chúng ta muốn biết số lượng bài viết theo tag . Chúng ta có thể làm như sau:

|  |
| --- |
| documentStore.DatabaseCommands.PutIndex(      "BlogPosts/PostsCountByTag",      new IndexDefinitionBuilder<BlogPost, BlogTagPostsCount> {          // Hàm Map: với mỗi tag của mỗi bài viết, tạo mới một đối tượng //BlogTagPostsCount với tên của tag và số lượng là 1          Map = posts => from post in posts                         from tag in post.Tags                         select new                         {                             Tag = tag,                             Count = 1                         },            // Hàm Reduce: nhóm tất cả đối tượng BlogTagPostsCount chúng ta //lấy được từ hàm Map, sử dụng tên của Tag như là khóa và tính tổng số //lượng mỗi tag. Bởi vì hàm Map gán giá trị Count cho mỗi Tag là 1 nên //khi hàm Reduce trả về chúng ta sẽ có đúng số lượng bài viết tương ứng //với mỗi Tag          Reduce = results => from result in results                              group result by result.Tag                                  into g                                  select new                                  {                                      Tag = g.Key,                                      Count = g.Sum(x => x.Count)                                  }      }); |

* + BlogTagPostsCount  được khai báo như sau:

|  |
| --- |
| public class BlogTagPostsCount  {      public string Tag { get; set; }      public int Count { get; set; }  } |

* + Một cách làm khác tốt hơn là tạo ra lớp index, sử dụng IndexCreation.CreateIndexes để đăng ký với server:

|  |
| --- |
| public class BlogPosts\_PostsCountByTag : AbstractIndexCreationTask<BlogPost, BlogPosts\_PostsCountByTag.ReduceResult>  {      public class ReduceResult      {          public string Tag { get; set; }          public int Count { get; set; }      }        // tên index được tạo ra sẽ là: BlogPosts/PostsCountByTag      public BlogPosts\_PostsCountByTag()      {          Map = posts => from post in posts                              from tag in post.Tags                              select new                                         {                                             Tag = tag,                                             Count = 1                                         };            Reduce = results => from result in results                                   group result by result.Tag                                   into g                                   select new                                              {                                                Tag = g.Key,                                                Count = g.Sum(x => x.Count)                                              };      }  } |

* + Chúng ta có thể để cho RavenDB tự quyết định sử dụng index nào hoặc có thể chỉ định cho RavenDB biết index nào được sử dụng trong khi truy vấn. Ví dụ dưới đây cho phép chúng ta tìm số lượng bài viết có tag là “RavenDB”:

|  |
| --- |
| // Truy vấn index thứ nhất mà chúng ta định nghĩa, sử dụng lớp  BlogTagPostsCount    var blogTagPostsCount = session.Query<BlogTagPostsCount>("BlogPosts/PostsCountByTag")      .FirstOrDefault(x => x.Tag == "RavenDB")      ?? new BlogTagPostsCount();  count = blogTagPostsCount.Count;    // Nếu sử dụng AbstractIndexCreationTask, chúng ta có thể sử dụng như sau  // Lưu ý là chúng ta sử dụng lớp ReduceResluts để lưu thông tin trả về  var tagPostsCount = session.Query<BlogPosts\_PostsCountByTag.ReduceResult, BlogPosts\_PostsCountByTag>()      .FirstOrDefault(x => x.Tag == "RavenDB")      ?? new BlogPosts\_PostsCountByTag.ReduceResult();  count = tagPostsCount.Count; |

###### Đánh chỉ mục dữ liệu theo cấp bậc

* Một đặc điểm nổi bật của document database là chúng ta rất ít bị giới hạn về cấu trúc dữ liệu. Một trong những trường hợp phổ biến là sử dụng cấu trúc dữ liệu theo cấp bậc. Một ví dụ thông dụng nhất là cấu trúc comment, như document bên dưới đây:

|  |
| --- |
| { //posts/123  'Name': 'Hello Raven',  'Comments': [  {  'Author': 'Ayende',  'Text': '...',  'Comments': [  {  'Author': 'Rahien',  'Text': '...',  "Comments": []  }  ]  }  ]  } |

* RavenDB hỗ trợ indexing (đánh chỉ mục) theo cấp bậc, chúng ta có thể định nghĩa index theo cú pháp bên dưới:

|  |
| --- |
| from post in docs.Posts  from comment in Hierarchy(post, "Comments")  select new { comment.Text } |

* Tất cả comment của bài viết sẽ được index theo tiến trình mà không quan tâm đến vị trí của nó trong hệ thống cấp bậc.

###### Quản lý lỗi trong indexes

* Indexes trong RavenDB được cung cấp truy vấn Linq chạy trên mô hình dữ liệu động JSON. Vì thế có rất nhiều lỗi có thể xảy ra, có thể vì định nghĩa index bị thay đổi hay bị mất dữ liệu trong các đối tượng JSON document.
* Trình biên tập lỗi index:
  + Một định nghĩa index sau sẽ bị lỗi:

|  |
| --- |
| { "Map" : "from doc in docs where doc.Type == 'posts' select new{ doc.Title.Length }" } |

* + Lỗi này là do sử dụng dấu nháy đơn để đánh dấu chuỗi, một số thứ như vậy không được chấp nhận trong C#. Điểu này sẽ dẫn đến lỗi trong trình biên tập lỗi như sau:

|  |
| --- |
| {         "url":"/indexes/PostsByTitle",         "error":"System.InvalidOperationException: Could not understand query: \r\n-- line 1 col 44: Char not terminated\r\n-- line 1 col 50: Char not terminated\r\n-- line 1 col 47:                            invalid QueryExpressionBody\r\n\r\n     at Raven.Database.Linq.QueryParsingUtils.GetVariableDeclaration(String query)"  } |
|  |

* + Lỗi này không rõ ràng lắm dưới địng dạng JSON, nó có nghĩa là:
    - -- line 1 col 44: Char not terminated
    - -- line 1 col 50: Char not terminated
    - -- line 1 col 47: invalid QueryExpressionBody
  + RavenDB sẽ đưa ra đầy đủ thông tin vì sao bị lỗi. Những lỗi này là ngay lập tức là không yêu cầu thêm thao tác nào tới cơ sở dữ liệu. Điều duy nhất mà người dùng làm là sửa lại định nghĩa index.
* Trình thực thi lỗi:
  + Một trường hợp phổ biến là một index không thể tác động đến tất cả documents tồn tại trên server. Ví dụ chúng ta có index sau:

|  |
| --- |
| { "Map" : "from doc in docs select new{ doc.Title.Length }" } |

* + Index này giả sử tất cả document đều có thuộc tính Title. Một document không có thuộc tính này sẽ trả về null khi truy cập đến nó. Kết quả báo NullReferenceException khi index được thực thi.
  + Bởi vì index được cập nhật với một tiến trình nền bên dưới nên người dùng sẽ không nhận biết được những lỗi này.
  + Trong RavenDB, lỗi index được thực hiện ở hai nơi. Đầu tiên là trong thống kê cơ sở dữ liệu (database statistics). Truy cập vào đó theo “/stats” là từ file chúng ta có thể đọc được tại ‘/raven/studio.html’ hoặc tại ‘/raven/statistics.html’.
  + Kết quả xuất ra của ‘/stats’ như sau:

|  |
| --- |
| {      "LastDocEtag": "00000000-0000-0b00-0000-000000000001",      "LastAttachmentEtag": "00000000-0000-0000-0000-000000000000",      "CountOfIndexes": 1,      "ApproximateTaskCount": 0,      "CountOfDocuments": 1,      "StaleIndexes": [],      "CurrentNumberOfItemsToIndexInSingleBatch": 512,      "CurrentNumberOfItemsToReduceInSingleBatch": 256,      "Indexes":[          {              "Name": "PostsByTitle",              "IndexingAttempts": 1,              "IndexingSuccesses": 0,              "IndexingErrors": 1          }      ],      "Errors":[          {              "Index": "PostsByTitle",              "Error": "Cannot   perform   runtime   binding   on   a   null   reference",              "Timestamp": "\/Date(1271778107096+0300)\/",              "Document": "bob"          }      ]  } |

* + Bộ sưu tập lỗi chứa 50 lỗi cuối cùng xảy ra trên server.
  + Bên cạnh đó, các bản ghi server (logs) chứa thêm các thông tin về lỗi.
* Vô hiệu hóa index:
  + Để bảo vệ cho RavenDB khỏi những index bị lỗi, RavenDB sẽ vô hiệu hóa các index lỗi này. Để vô hiệu hóa index thì:
    - Nếu một index có tỉ lệ lỗi từ 15% trở lên – nó sẽ bị vô hiệu hóa.
    - 15% này được tính sau khi đã qua 10 documents đầu tiên (tránh trường hợp vô hiệu hóa index ngay lập tức nếu document đầu tiên trong index không hợp lệ)
  + Một index bị vô hiệu hóa thì không thể truy vấn được, tất cả truy vấn tới index bị hiệu hóa sẽ dẫn đến phát sinh lỗi như sau:

|  |
| --- |
| {           "url":"/indexes/PostsByTitle",           "error":"Index   PostsByTitle   is   invalid,   out   of   10   indexing   attempts,   10   has   failed.\r\nError   rate   of   100%   exceeds   allowed   15%   error   rate",           "index":"PostsByTitle"  } |

* + Điều duy nhất có thể làm với những index này là xoa chúng đi hoặc thay thế định nghĩa index mới.

###### Quản lý mối quan hệ giữa các document

###### Tính đa hình của indexes

* Mặc định trong RavenDB, index chỉ được thực hiện trên các thực thể cụ thể hoặc là trên Collection và nó bỏ qua hệ thống phân cấp kế thừa.
* Ví dụ chúng ta có hệ thống phân cấp kế thừa như sau:



* Nếu chúng ta lưu trữ một đối tượng Cat thì chúng ta sẽ có thực thể thuộc nhóm “Cats” và khi ta lưu trữ một đối tượng Dog thì chúng ta sẽ có thực thể thuộc nhóm “Dogs”.
* Nếu chúng ta muốn có index truy vấn thực thể Cats theo tên thì ta sẽ viết:

|  |
| --- |
| from cat in docs.Cats  select new { cat.Name } |

Tương tự cho Dogs

|  |
| --- |
| from dog in docs.Dogs  select new { dog.Name } |

* Những index này chỉ cho chúng ta kết quả là những động vật đã được xác định trên. Nhưng nếu muốn truy vấn trên tất cả các loài vật thì chúng ta sẽ truy vấn như thế nào?
* Multi-map index:
  + Cách đơn giản nhất để giải quyết vấn đề này là làm như sau:

|  |
| --- |
| public class AnimalsIndex : AbstractMultiMapIndexCreationTask  {      public AnimalsIndex()      {          AddMap<Cat>(cats => from c in cats                              select new { c.Name });            AddMap<Dog>(dogs => from d in dogs                              select new { d.Name });      }  } |

* + Truy vấn như sau:

|  |
| --- |
| var results = session.Advanced.LuceneQuery<object>("AnimalsIndex").WhereEquals("Name", "Mitzy"); |

* + Cũng có thể sử dụng cung cấp Linq nếu đối tượng triển khai từ interface, thực thể IAnimal:

|  |
| --- |
| session.Query<IAnimal>("AnimalsIndex").Where(x => x.Name == "Mitzy"); |

* Cách khác:
  + Thay đổi cách chúng ta tạo ra tên thực thể của lớp con của lớp Animal:

|  |
| --- |
| var documentStore = new DocumentStore()  {      Conventions =      {          FindTypeTagName = type =>          {              if (typeof(Animal).IsAssignableFrom(type))                  return "Animals";              return DocumentConvention.DefaultTypeTagName(type);          }      }  }; |

* + Sử dụng phương thức này, chúng ta có thể định nghĩa index để truy vấn trên tất cả động vật:

|  |
| --- |
| from animal in docs.Animals  select new { animal.Name } |

* + Nhưng sẽ ra sao nếu chúng ta không muốn thay đổi tên thực thể? Chúng ta sẽ định nghĩa một index đa hình:

|  |
| --- |
| from animal in docs.WhereEntityIs("Cats", "Dogs")   select new { animal.Name } |

* + Điều này sẽ tạo ra index phù hợp với cả Cats và Dogs.

#### Làm việc bất đồng bộ

* RavenDB client API hỗ trợ thực hiện các thao tác bất đồng bộ, vì thế những thao tác quá dài sẽ không bị khóa. Điều nãy cũng cho thấy RavenDB SilverLight client làm việc như thế nào.
* Hỗ trợ System.Threading.Tasks

|  |
| --- |
| var entity = new Company { Name = "Async Company #2", Id = "companies/2" };  using (var session = documentStore.OpenAsyncSession())  {      var company = session.LoadAsync<Company>(1); // loading an entity asynchronously        session.Store(entity); // in-memory operations are committed asynchronously when calling SaveChangesAsync      session.SaveChangesAsync(); // returns a task that completes asynchronously        var query = session.Query<Company>()          .Where(x => x.Name == "Async Company #1")          .ToListAsync(); // returns a task that will execute the query  } |

#### Partial document updates sử dụng Pacthing API

* Quá trình cập nhật document cho phép thay đổi document trên server mà không cần phải tải về đầy đủ document đó và lưu document trở lại server. Việc này hữu ích trong việc cập nhật dữ liệu phi chuẩn hóa (denormalized data) vào các thực thể.
* Thông thường, máy khách sẽ phát ra lệnh Load đến server, deserialize kết quả vào thực thể, thực hiện các thay đổi cho thực thể đó và sau đó gửi nó lại cho server serialize. Sử dụng Patching API, máy khách có thể phát hành một lệnh Patch duy nhất và máy chủ sẽ thực hiện các thao tác được yêu cầu vào biểu diễn JSON của document. Điều này sẽ tiết kiệm được băng thông và thực hiện nhanh hơn nhưng không phải là một transaction nên chỉ có lệnh patching cuối cùng mới được tồn tại.
* Lưu ý: Vì tính năng này liên quan đến thao tác cấp thấp đối với document, nó được coi là tính năng chuyên gia và nói chung không nên sử dụng như một giải pháp mục đích chung. Nếu chúng ta gặp trường hợp mà cần phải sử dụng phương pháp này thì nên kiểm tra lại mô hình dữ liệu và xem thử có thể tối ưu hóa để ngăn chặn việc sử dụng các Patching API. Ngoại lệ duy nhất là cập nhật dữ liệu phi chuẩn hóa, nơi phương pháp này là hợp lệ nhưng không phải lúc nào cũng được đề xuất.
* Patching API được biểu diễn thông qua RavenDB’s DatabaseCommand, có sẵn từ đối tượng document store và session.Advanced. Một lệnh Patch được thực hiện bằng cách gọi một hàm Patch() duy nhất, chấp nhận 3 tham số: document key, một mảng của PatchRequest và tùy chọn Etag:

|  |
| --- |
| var comment = new BlogComment                  {                      Title = "Foo",                      Content = "Bar"                  };    documentStore.DatabaseCommands.Patch(      "blogposts/1234",      new[]          {              new PatchRequest                  {                      Type = PatchCommandType.Add,                      Name = "Comments",                      Value = RavenJObject.FromObject(comment)                  }          }); |

* Document key là key duy nhất của document trong database hiện tại, nơi lệnh patch sẽ thực hiện trên đó. Xác định một Etag sẽ đảm bảo những thay đổi chỉ được thực hiện nếu không có lệnh viết được thực hiện từ khi máy khách lấy được Etag chỉ định đó.
* Mô tả của đối tượng PatchRequest và các tùy chọn có sẵn thông qua các Patching API. Chúng ta sẽ sử dụng ví dụ đơn giản về công cụ blog, sử dụng những lớp sau:

|  |
| --- |
| public class BlogPost  {      public string Id { get; set; }      public string Title { get; set; }      public string Category { get; set; }      public string Content { get; set; }      public DateTime PublishedAt { get; set; }      public string[] Tags { get; set; }      public BlogComment[] Comments { get; set; }  }  public class BlogComment  {      public string Title { get; set; }      public string Content { get; set; }  } |

* Đối tượng PatchRequest:
  + Khi tạo đối tượng PatchRequest để sử dụng lệnh patch, ít nhất 2 thuộc tính được chỉ định: Name và Type.
  + Name là đường dẫn từ gốc đến thuộc tính đó trong đối tượng. Cú pháp giống với XPath thực hiện trên XML, hoặc đơn giản hơn.
  + Type dùng để xác định lệnh Patch. Có thể là một trong số loại dưới đây:
    - **Set** - Set a property
    - **Unset** - Unset (remove) a property
    - **Inc** - Increment a property by a specified value
    - **Rename** - Rename a property
    - **Copy** - Copy a property value to another property
    - **Modify** - Modify a property value by providing a nested set of patch operation
    - **Add** - Add an item to an array
    - **Insert** - Insert an item to an array at a specified position
    - **Remove** - Remove an item from an array at a specified position
* Thực hiện một lệnh cập nhật đơn giản:
  + Một thuộc tính trong document là một trường dữ liệu trong thực thể. Để thay đổi giá trị của nó bằng cách sử dụng Patching API, cung cấp đường dẫn( path) trong giá trị Name và khởi tạo Type với PatchCommandType.Set . Sau đó, serialize đối tượng chúng ta muốn lưu vào thuộc tính đó và gửi vào cho nó một Value.
  + Giá trị mới chúng ta muốn thiết lập có thể là: kiểu dữ liệu truyền thống, một đối tượng hay là một tập hợp đối tượng. Sử dụng RavenJObject.FromObject(object) để serialize một cách dễ dàng:

|  |
| --- |
| // Setting a native type value  documentStore.DatabaseCommands.Patch(      "blogposts/1234",      new[]          {              new PatchRequest                  {                      Type = PatchCommandType.Set,                      Name = "Title",                      Value = RavenJObject.FromObject("New title")                  }          });    // Setting an object as a property value  documentStore.DatabaseCommands.Patch(      "blogposts/4321",      new[]          {              new PatchRequest                  {                      Type = PatchCommandType.Set,                      Name = "Author",                      Value = RavenJObject.FromObject(                          new BlogAuthor                              {                                  Name = "Itamar",                                  ImageUrl = "/author\_images/itamar.jpg"                              })                  }          }); |

* + Xóa một thuộc tính bằng cách thay đổi Type thành PatchCommandType.Unset.
  + Để đổi tên hay sao chép thuộc tính sang thuộc tính khác, chi định một đường dẫn mới vào Value:

|  |
| --- |
| // This is how you rename a property; copying works  // exactly the same, but with Type = PatchCommandType.Copy  documentStore.DatabaseCommands.Patch(      "blogposts/1234",      new[]          {              new PatchRequest                  {                      Type = PatchCommandType.Rename,                      Name = "Comments",                      Value = new RavenJValue("cmts")                  }          }); |

* + Giá trị số sử dụng như số đếm có thể tăng hoặc giảm mà không cần lo lắng về giá trị thực của nó. Sử dùng giá trị dương để tang và giá trị âm để giảm:

|  |
| --- |
| // Assuming we have a Views counter in our entity  documentStore.DatabaseCommands.Patch(      "blogposts/1234",      new[]          {              new PatchRequest                  {                      Type = PatchCommandType.Inc,                      Name = "Views",                      Value = new RavenJValue(1)                  }          }); |

* Điều kiện cập nhật:
  + Nếu PreVal được thiết lập, giá trị sẽ được so sánh với giá trị hiện tại của thuộc tính để xác định sự thay đổi không ghi đè lên giá trị mới. Nếu giá trị là null thì thao tác luôn thành công.
* Làm việc với mảng dữ liệu:
  + Bất kỳ collection nào trong thực thể sẽ được serialize thành mảng trong kết quả JSON document. Thực hiện thao tác trên tập hợp dễ dàng bằng cách sử dụng thuộc tính Position:

|  |
| --- |
| // Append a new comment; Insert operation is supported  // as well, by using PatchCommandType.Add and  // specifying a Position to insert at  documentStore.DatabaseCommands.Patch(      "blogposts/1234",      new[]          {              new PatchRequest                  {                      Type = PatchCommandType.Add,                      Name = "Comments",                      Value =                          RavenJObject.FromObject(new BlogComment                                                      {Content = "FooBar"})                  }          });    // Remove the first comment  documentStore.DatabaseCommands.Patch(      "blogposts/1234",      new[]          {              new PatchRequest                  {                      Type = PatchCommandType.Remove,                      Name = "Comments",                      Position = 0                  }          }); |

* Làm việc với thao tác lồng nhau (nested operations):
  + Thao tác lồng nhau hợp lệ với Type là PatchCommandType.Modify
  + Nếu muốn thay đổi tất cả phần tử trong collection chúng ta có thể thiết lập thuộc tính AllPosition thành “true”.
  + Một số ví dụ về thao tác lồng nhau:

|  |
| --- |
| // Set value in a nested element:  var addToPatchedDoc = new JsonPatcher(doc).Apply(      new[]  {      new PatchRequest      {          Type = PatchCommandType.Modify,          Name = "user",          Nested = new[]          {              new PatchRequest {Type = PatchCommandType.Set, Name = "name", Value = new RavenJValue("rahien")},          }      },  }); |

|  |
| --- |
| // Remove value in a nested element:  var removeFromPatchedDoc = new JsonPatcher(doc).Apply(  new[]  {      new PatchRequest      {          Type = PatchCommandType.Modify,          Name = "user",          PrevVal = RavenJObject.Parse(@"{ ""name"": ""ayende"", ""id"": 13}"),          Nested = new[]          {              new PatchRequest {Type = PatchCommandType.Unset, Name = "name" },          }      },  }); |

#### Set based Operations (Cài đặt nhiều thao tác cùng lúc)

* Nhiều lúc chúng ta cần cập nhật hay xóa một số lượng lớn document thỏa mãn điều kiện nào đấy. Với SQL chúng ta có thể làm được tương tự như sau:

|  |
| --- |
| DELETE FROM Users WHERE LastLogin < '2009-01-01'  UPDATE Users SET IsActive = 0 WHERE LastLogin < '2010-01-01' |

* Chúng ta không thể thực hiện điều này trong document database vì một tập hợp các thao tác cơ sở dữ liệu không được hỗ trợ. RavenDB có hỗ trợ điều này bằng cách đưa vào một truy vấn và định nghĩa một thao tác cơ sở dữ liệu. Nó sẽ chạy câu truy vấn và thực hiện thao tác cơ sở dữ liệu dựa trên kết quả của nó.
* Các truy vấn và index giống nhau sử dụng để truy xuất dữ liệu được dùng cho tập hợp thao tác dữ liệu. Do đó cú pháp để xác định những document nào được thực hiện cũng tương tự như cách chúng ta lấy dữ liệu những document này lên từ cơ sở dữ liệu.
* Thao tác xóa:
  + Để thực hiện lệnh xóa cùng lúc nhiều document thì ta cần chỉ định một index và một truy vấn được gửi đến index này. Để giảm thiểu khả năng nhận được các kết quả cũ thì chúng ta nên thực hiện trên static index:

|  |
| --- |
| documentStore.DatabaseCommands.DeleteByIndex("IndexName",      new IndexQuery      {  Query = "Title:RavenDB" // entity.Title chứa RavenDB      }, allowStale: false); |

* Thao tác cập nhật (Patching API):
  + Thực hiện cập nhật nhiều document tương tự như xóa nhiều documents. Chúng ta sử dụng patching API để dễ dàng thực hiện thao tác với những documents thỏa mãn điều kiện.

|  |
| --- |
| documentStore.DatabaseCommands.UpdateByIndex("IndexName",      new IndexQuery {Query = "Title:RavenDB"},      new[]      {         new PatchRequest         {          Type = PatchCommandType.Add,             Name = "Comments",             Value = "New automatic comment we added programmatically"         }       }, allowStale: false); |

* Vấn đề stale index:
  + Mặc định, set based operations không thực hiện trên những stale index. Thao tác này chỉ thực hiện thành công nếu index được chỉ định không phải stale index. Điều này đảm bảo rằng chúng ta chỉ xóa những gì chúng ta thực sự muốn xóa và nó là một phần của RavenDB’s safe by default.
  + Đối với những index mà việc cập nhật dữ liệu thường xuyên được thực hiện, chúng ta có thể thiết lập một Cutoff (điểm ngắt) trong đối tượng IndexQuery để đảm bảo rằng thao tác này được thực hiện và kết quả là kết quả chúng ta mong đợi.
  + Khi chúng ta chắc chắn thực hiện được thao tác này trên stale index, chỉ cần thiết lập tham số allowStale thành “true”.

### Tổng quan HTTP API

* RavenDB hỗ trợ HTTP API cho việc truy cập và thao tác dữ liệu trên máy chủ. HTTP API cung cấp hầu hết các chức năng tương tự C# .NET client API, nhưng với platform agnostic (tạm dịch là đa nền tảng) và giao diện web thân thiện. Sử dụng HTTP API chúng ta có thể viết được ứng dụng RavenDB với đầy đủ chức năng chỉ cần sử dụng Javascript và HTML.
* Là một phần của web thân thiện, HTTP API hiêu được những nguyên tắc chung RESTful. Ví dụ, document database là những tài nguyên thông qua những địa chỉ URLs duy nhất và những nguồn tài nguyên có thể thực thi bằng cách sử dụng các động từ đặc trưng của HTTP như: GET, PUT, POST và DELETE.
* Tuy nhiên, trong khi RESTful là mục đích của HTTP API

# CHƯƠNG 4: XÂY DỰNG ỨNG DỤNG SỬ DỤNG RAVENDB

## Giới thiệu về ứng dụng

## Phân tích, thiết kế hệ thống

## Triển khai

## Test performance để thể hiện sức mạnh của NoSQL

# CHƯƠNG 5: TỔNG KẾT

## Kết quả đạt được

### Về mặt lý thuyết

### Về mặt thực nghiệm

## Hướng phát triển

# Tài liệu tham khảo