**ĐẠI HỌC QUỐC GIA TP. HỒ CHÍ MINH**

**TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN**

**KHOA CÔNG NGHỆ PHẦN MỀM**

****

**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 12 năm 2012***

**LỜI MỞ ĐẦU**

Ngày nay với kỹ nguyên công nghệ bùng nổ, thành công của Internet đã khiến cho số lượng người dùng truy cập vào cùng một hệ thống ngày càng tăng. Điển hình như Facebook một tháng phục vụ hơn 1000 tỉ truy cập và hơn 800 triệu lượt khách ghé thăm thì ta mới hình dung được sự bùng nổ của thông tin như thế nào. Để giải quyết vấn đề bùng nổ như trên thì chúng ta đã mở rộng các hệ thống máy chủ siêu lớn, phân thành nhiều cụm đặt khắp nơi trên thế giới. Nhưng với tốc độ phát triển theo cấp số như hiện nay thì việc tăng số lượng máy chủ thôi vẫn chưa đủ. Ta cần xem xét và nâng cấp các giải pháp lưu trữ cho tương lai.

Hệ thống máy chủ cơ sở dữ liệu đòi hỏi phải rất mạnh mẽ nếu không máy chủ sẽ bị quá tải. Với các hệ thống với số lượng lên đến hàng triệu cho đến hàng tỉ thì việc hiệu năng tốt là việc bắt buộc.Các hệ RBDMs hiện nay thì vấn đề hiệu năng thường không tốt cho trường hợp này. Ngôn ngữ SQL là ngôn ngữ thông dịch với các ràng buộc trong các bảng khiến cho hiệu năng thực sự của hệ thống cơ sở dữ liệu khi thực thi là khá ì ạch với hệ thống lớn như kể trên.Chưa kể là với hệ thống lớn thì vấn đề phân tán dữ liệu, tính toàn vẹn dữ liệu là việc rất quan trọng. NoSQL đáp ứng được tất cả các yêu cầu này.Với tốc độ nhanh do không phải qua các câu truy vấn SQL, có tính sẵn sàng, phân tán cao và độ ổn định tuyệt vời. Rất thích hợp cho các hệ thống có lượt truy vấn lớn. Ở trong khoá luận này, chúng tôi sẽ nghiên cứu về một loại NoSQL khá phổ biến – RavenDB.

RavenDB là một cơ sở dữ liệu 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. RavenDB xứng đáng là một cơ sở dữ liệu đáng tin cậy.

**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 Công nghệ 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 12, tháng 12, 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

DANH MỤC CÁC BẢNG, SƠ ĐỒ……………………………………………….......1

DANH MỤC CÁC HÌNH……………………………………………………………...1

[1 CHƯƠNG 1 – Giới thiệu đề tài 1](#_Toc344307479)

[1.1 Vấn đề tìm hiểu 1](#_Toc344307480)

[1.2 Mục tiêu đề tài 2](#_Toc344307481)

[2 CHƯƠNG 2 – Tổng quan về cơ sở dữ liệu NoSQL 3](#_Toc344307482)

[2.1 NoSQL là gì ? 3](#_Toc344307483)

[2.2 Một số khái niệm mới NoSQL: 3](#_Toc344307484)

[2.3 Lịch sử 4](#_Toc344307485)

[2.4 Định nghĩa 4](#_Toc344307486)

[2.5 Kiến trúc 4](#_Toc344307487)

[2.6 Một số thuật ngữ liên quan 5](#_Toc344307488)

[2.7 Ưu nhược điểm của cơ sở dữ liệu NoSQL: 6](#_Toc344307489)

[2.7.1 Ưu điểm: 6](#_Toc344307490)

[2.7.2 Nhược điểm: 7](#_Toc344307491)

[2.8 Cách triển khai một ứng dụng NoSQL 8](#_Toc344307492)

[2.8.1 Xác định NoSQL có phù hợp 9](#_Toc344307493)

[2.8.2 Thiết kế cấu trúc dữ liệu dạng document 9](#_Toc344307494)

[2.8.3 Ví dụ so sánh cách thiết kế mô hình dữ liệu trong RDMBs và Document Database: 11](#_Toc344307495)

[3 CHƯƠNG 3: TÌM HIỂU CÁC GIẢI PHÁP CƠ SỞ DỮ LIỆU NOSQL 20](#_Toc344307496)

[3.1 Key-Value Store / Tuple Store 20](#_Toc344307497)

[3.2 Wide Column Store / Column Families 20](#_Toc344307498)

[3.3 Document Store 26](#_Toc344307499)

[3.3.1 Giới thiệu 26](#_Toc344307500)

[3.3.2 Document (tài liệu) 27](#_Toc344307501)

[3.3.3 Keys (Khóa) 27](#_Toc344307502)

[3.3.4 Retrieval (truy lại hay tìm kiếm thông tin) 27](#_Toc344307503)

[3.3.5 Organization (Tổ chức) 28](#_Toc344307504)

[3.3.6 Implementations 28](#_Toc344307505)

[3.4 Graph Database 31](#_Toc344307506)

[4 CHƯƠNG 4: TÌM HIỂU VỀ RAVENDB 33](#_Toc344307507)

[4.1 Khái niệm cơ bản về RavenDB 33](#_Toc344307508)

[4.2 Tại sao chọn RavenDB 33](#_Toc344307509)

[4.3 Tính năng của RavenDB 35](#_Toc344307510)

[4.4 Cơ sở lý thuyết RavenDB 41](#_Toc344307511)

[4.4.1 Một số thông tin cần biết về RavenDB 41](#_Toc344307512)

[4.4.2 Hướng dẫn thêm RavenDB vào ứng dụng nhanh chóng 43](#_Toc344307513)

[4.4.3 Yêu cầu hệ thống 46](#_Toc344307514)

[4.4.4 Lý thuyết cơ bản RavenDB 46](#_Toc344307515)

[4.4.5 .NET client API 50](#_Toc344307516)

[4.4.6 Tổng quan HTTP API 90](#_Toc344307517)

[4.4.7 Phía máy chủ RavenDB 90](#_Toc344307518)

[5 CHƯƠNG 5: XÂY DỰNG ỨNG DỤNG SỬ DỤNG RAVENDB 98](#_Toc344307519)

[5.1 Giới thiệu về ứng dụng 98](#_Toc344307520)

[5.2 Lý do lựa chọn ứng dụng này 98](#_Toc344307521)

[5.3 Phân rã chức năng website 98](#_Toc344307522)

[5.4 Ý tưởng thiết kế 100](#_Toc344307523)

[5.4.1 Thiết kế mô hình 3 tầng (3 Tier): Clients – Web server – Database server 100](#_Toc344307524)

[5.4.2 Kiến trúc Website 101](#_Toc344307525)

[5.5 Tổng quan ASP.NET MVC4 102](#_Toc344307526)

[5.5.1 Khái niệm 102](#_Toc344307527)

[5.5.2 Lịch sử phát triển. 103](#_Toc344307528)

[5.5.3 Những tính năng mới 103](#_Toc344307529)

[5.6 Phân tích, thiết kế hệ thống 104](#_Toc344307530)

[5.6.1 Class diagram 104](#_Toc344307531)

[5.6.2 Sequence diagram 105](#_Toc344307532)

[5.7 Thiết kế giao diện 112](#_Toc344307533)

[5.8 Test performance để thể hiện sức mạnh của NoSQL 114](#_Toc344307534)

[6 CHƯƠNG 6: TỔNG KẾT 115](#_Toc344307535)

[6.1 Kết quả đạt được 115](#_Toc344307536)

[6.1.1 Về mặt lý thuyết 115](#_Toc344307537)

[6.1.2 Về mặt thực nghiệm 115](#_Toc344307538)

[6.2 Hướng phát triển 115](#_Toc344307539)

TÀI LIỆU THAM KHẢO

**DANH MỤC CÁC BẢNG, SƠ ĐỒ**

**DANH MỤC CÁC HÌNH**

# CHƯƠNG 1 – Giới thiệu đề tài

## Vấn đề tìm hiểu

Trong khoảng hơn 2 thập niên trở lại đây, hệ quản trị cơ sở dữ liệu quan hệ - RDBMs là sự lựa chọn duy nhất cho việc quản trị cơ sở dữ liệu. Tuy nhiên, với các yêu cầu mới hiện nay thì RDBMs đã bộc lộ yếu điểm. Chính sự quá chặc chẽ, yêu cầu nhất quán dữ liệu đã gây ra sự rườm rà, phức tạp làm giảm hiệu xuất hoạt động, nhất là trong trường hợp phải chứa một lượng lớn dữ liệu. Nhưng với sự bùng nổ công nghệ như hiện nay, nhất là với mạng Internet thì lượng dữ liệu cần lưu trữ ngày càng tăng. Yêu cầu cho việc lưu trữ ngày càng cao như: lưu trữ nhiều dữ liệu, tốc độ truy xuất nhanh, phân tán dữ liệu trên nhiều máy chủ… thì với mô hình cơ sở dữ liệu quan hệ như hiện nay thì rõ ràng không thể đáp ứng đủ các yêu cầu trên.

Mọi vấn đề đều có giải pháp. Thật vậy, những năm gần đây đã nổi lên một xu hướng lưu trữ mới, một cách thức trái ngược với cơ sở dữ liệu quan hệ - đó là cơ sở dữ liệu phi quan hệ - NoSQL. NoSQL sinh ra để khắc phục các vấn đề mà một cơ sở dữ liệu dạng RDBMs gặp phải. NoSQL sinh ra không phải để cạnh tranh với RDBMs mà là để đảm nhiệm những việc mà RDBMs chưa làm tốt.

Mục tiêu mà NoSQL nhắm đến đó là hiệu suất hoạt động cao với số lượng dữ liệu cực lớn. Tuy nhiên để đạt được điều đó thì NoSQL đã bỏ qua thông dịch trong SQL cùng với những truy vấn rườm ra. Việc sử dụng các ràng buộc quan hệ cùng truy vấn SQL có vẻ thân thiện và thích hợp với phần đông dữ liệu. Tuy nhiên, nếu dữ liệu quá đơn giản, các thủ tục SQL sẽ không cần thiết (theo Curt Monash - một nhà phân tích cơ sở dữ liệu, một blogger). Đồng thời NoSQL cũng có cách thiết kế dữ liệu khác với cơ sở dữ liệu truyền thống như: tư tưởng thiết kế dữ liệu phi quan hệ, lưu trữ dữ liệu dạng document, sử dụng tối đa indexes… Trong các đặc tính đó, dữ liệu phi quan hệ là một yếu tố quan trọng góp phần làm thanh công cho NoSQL. Dữ liệu phi quan hệ tức là không tuân theo các dạng chuẩn hóa mà cơ sở dữ liệu RDBMs đặt ra. Thay vào đó, khi thiết kế một cơ sở dữ kiệu NoSQL ta phải tuân theo một số quy tắc mới mà NoSQL đặt ra để đạt được hiệu suất hoạt động cao.

Bảng dưới đây chỉ ra kết quả làm việc trên MySQL và cơ sở dữ liệu Cassandra của Facebook.

|  |  |  |
| --- | --- | --- |
| Facebook Search  > 50 GB Data | | |
|  | **MySQL** | **Cassandra** |
| Writes Average | ~300ms | 0.12ms |
| Reads Average | ~350ms | 15ms |

*Hiệu suất hoạt động trên MySQL và Cassandra*

Chính sự khác biệt giữa 2 loại cơ sở dữ liệu này dẫn đến cách thiết kế cũng khác nhau. Đa số các lập trình viên đều quen với mô hình quan hệ truyền thống, do đó cần phải tìm hiểu kĩ cách thiết kế dữ liệu của NoSQL để đạt được hiệu suất mong muốn.

Đồng ý rằng RDBMs cung cấp một mô hình tuyệt vời để đảm bảo tính toàn vẹn dữ liệu. Tuy nhiên, rất nhiều người lựa chọn NoSQL đã nói rằng chúng không quá cần thiết cho nhu cầu của họ.

Như vậy, trong đề tài này chúng tôi sẽ tìm hiểu xem NoSQL đã giải quyết các vấn đề trên như thế nào và áp dụng kiến thức tìm hiểu đó vào việc xây dựng một ứng dụng sử dụng cơ sở dữ liệu dạng NoSQL.

## Mục tiêu đề tài

* Tìm hiểu NoSQL, kiến trúc, phân loại và đặc điểm của từng loại để có cái nhìn tổng quan về NoSQL đồng thời biết được cách mà NoSQL đã giải quyết được vấn đề hiệu suất cao với lượng dữ liệu lớn như thế nào.
* Tìm hiểu trường hợp áp dụng cơ sở dữ liệu dạng NoSQL, trường hợp nào không phù hợp với NoSQL. Phân tích làm rõ ưu khuyết điểm của việc áp dụng cơ sở dữ liệu NoSQL. So sánh giữa việc sử dụng cơ sở dữ liệu RDBMs và cơ sở dữ liệu NoSQL trên cùng một ứng dụng.
* So sánh hiệu suất giữa một cơ sở dữ liệu dạng NoSQL và cơ sở dữ liệu dạng RDBMs để làm rõ hiệu suất hoạt động của NoSQL.
* Tìm hiểu cách thiết kế một cơ sở dữ liệu cho ứng dụng sử dụng NoSQL.
* Tìm hiểu các triển khai NoSQL phổ biến như: RavenDB, MongoDB, CouchDB, MemCache, Cassandra…, ưu khuyết điểm của từng loại, tiêu chí để lựa chọn một triển khai cho ứng dụng thực tế.
* Sử dụng các kiến thức về NoSQL để xây dựng một ứng dụng sử dụng cơ sở dữ liệu dạng NoSQL đồng thời để tổng hợp lại kiến thức đã học trước đây. Ở đây chúng tôi quyết định xây dựng một website cho phép các người dùng có thể thảo luận về vấn đề nào đó (với các chức năng cơ bản như Google Group) bởi vì ứng dụng có các tính chất phụ hợp với cơ sở dữ liệu dạng NoSQL. Đồng thời sử dụng RavenDB làm cơ sở dữ liệu cho website này bởi trong loại cở sở dữ liệu dạng Document thì RavenDB có rất nhiều ưu điểm và sẽ được làm rỏ hơn trong Chương 4 Tìm hiểu về RavenDB.

# CHƯƠNG 2 – Tổng quan về cơ sở dữ liệu NoSQL

## NoSQL là gì ?

NoSQL còn có nghĩa là Non-Relational (NoRel) - không ràng buộc. Tuy nhiên, thuật ngữ đó ít phổ biến hơn và ngày nay người ta thường dịch NoSQL thành Not Only SQL - Không chỉ SQL. NoSQL ám chỉ đến những cơ sở dữ liệu không dùng mô hình dữ liệu quan hệ để quản lý dữ liệu trong lĩnh vực phần mềm, bởi thế nếu dữ liệu của bạn thực sự có tính quan hệ chặt chẽ, hãy sử dụng RDBMs.

NoSQL có dạng mô hình/lược đồ tự do (schema-free). Bạn không phải thiết kế các table và các cấu trúc như với SQL, việc duy nhất bạn phải làm là cứ insert các giá trị mới.

## Định nghĩa

Thế hệ database kế tiếp là một thế hệ cơ sở dữ liệu non-relational (không ràng buộc), distributed (phân tán), open source, horizontal scalable (khả năng mở rộng theo chiều ngang) có thể lưu trữ, xử lý từ một lượng rất nhỏ cho tới hàng petabytes dữ liệu trong hệ thống có độ chịu tải, lỗi cao với những đòi hỏi về tài nguyên phần cứng thấp.

Một số đặc điểm nhận dạng cho thế hệ database mới này bao gồm:

* Schema-free.
* Hỗ trợ mở rộng dễ dàng.
* API đơn giản.
* Eventual consistency (nhất quán cuối) và transactions hạn chế trên các thành phần dữ liệu đơn lẻ.
* Không giới hạn không gian dữ liệu…

NoSQL storage đặc biệt phổ dụng trong thời kỳ Web 2.0 bùng nổ, nơi các mạng dịch vụ dữ liệu cộng đồng cho phép người dùng tạo hàng tỷ nội dung trên web. Do đó, dữ liệu lớn rất nhanh vượt qua giới hạn phần cứng và cần phải giải quyết bằng bài toán phân tán. Nửa đầu năm 2009, người ta đã manh nha thuật ngữ NoSQL đánh dấu sự trưởng thành của thế hệ database mới trong khi những sản phẩm phần mềm có thể đã được phát triển từ trước đó rất lâu.

## Lịch sử

Thuật ngữ NoSQL được giới thiệu lần đầu vào năm 1998 sử dụng làm tên gọi chung cho các lightweight open source relational database (cơ sở dữ liệu quan hệ nguồn mở nhỏ) nhưng không sử dụng SQL cho việc truy vấn.

Vào năm 2009, Eric Evans, nhân viên của Rackspace giới thiệu lại thuật ngữ NoSQL trong một hội thảo về cơ sở dữ liệu nguồn mở phân tán. Thuật ngữ NoSQL đánh dấu bước phát triển của  thế hệ database mới: distributed (phân tán) + non-relational (không ràng buộc).

## Một số khái niệm trong NoSQL:

* **Fields**: tương đương với khái niệm Columns trong SQL
* **Document**: thay thế khái niệm row trong SQL. Đây cũng chính là khái niệm làm nên sự khác biệt giữa NoSQL và SQL, 1 document chứa số cột (fields) không cố định trong khi 1 row thì số cột(columns) là định sẵn trước.
* **Collection**: tương đương với khái niệm table trong SQL. Một collection là tập hợp các document. Điều đặc biệt là một collection có thể chứa các document hoàn toàn khác nhau.
* **Key-value**: cặp khóa - giá trị được dùng để lưu trữ dữ liệu trong NoSQL
* **Cursor**: tạm dịch là con trỏ. Chúng ta sẽ sử dụng cursor để lấy dữ liệu từ database.

**T**rong các hệ cơ sở dữ liệu quan hệ, các cột được định nghĩa theo bảng còn với hệ cơ sở  dữ liệu không ràng buộc, các cột được định nghĩa ở mỗi document. Bởi thế, các document quản lí gần như tất cả, các collection không cần quản lí chặt chẽ những gì đang xảy ra trong nó nữa.

Bảng tương quan giữa RDBMS và NoSQL:

|  |  |
| --- | --- |
| RDBMS | NoSQL |
| Columns | Fields |
| Row | Document |
| Table | Collection |
| Query: SQL | Query: using API |
| Foreign keys | Non Foreign keys |
| Schema | Free schema |

## Kiến trúc

Các RDBMs hiện tại đã bộc lộ những yếu kém như việc đánh chỉ mục một lượng lớn dữ liệu, phân trang, hoặc phân phối luồng dữ liệu media (phim, ảnh, nhạc, ...). Cơ sở dữ liệu quan hệ được thiết kế cho những mô hình dữ liệu nhỏ thường xuyên đọc viết trong khi các Social Network Services lại có một lượng dữ liệu cực lớn và cập nhật liên tục do số lượng người dùng quá nhiều ở một thời điểm. Thiết kế trên Distributed NoSQL giảm thiểu tối đa các phép tính toán, I/O liên quan kết hợp với batch processing đủ đảm bảo được yêu cầu xử lý dữ liệu của các mạng dịch vụ dữ liệu cộng đồng này. Facebook, Amazon là những ví dụ điển hình.

Về cơ bản, các thiết kế của NoSQL lựa chọn mô hình lưu trữ tập dữ liệu theo cặp giá trị key-value. Khái niệm node được sử dụng trong quản lý dữ liệu phân tán. Với các hệ thống phân tán, việc lưu trữ chấp nhận trùng lặp dữ liệu. Một request truy vấn tới data có thể gửi tới nhiều máy cùng lúc, khi một máy nào nó bị chết cũng không ảnh hưởng nhiều tới toàn bộ hệ thống. Để đảm bảo tính real time trong các hệ thống xử lý lượng lớn dữ liệu, thông thường người ta sẽ tách biệt database ra làm 2 hoặc nhiều database. Một database nhỏ đảm bảo vào ra liên tục, khi đạt tới ngưỡng thời gian hoặc dung lượng, database nhỏ sẽ được gộp (merge) vào database lớn có thiết kế tối ưu cho phép đọc (read operation). Mô hình đó cho phép tăng cường hiệu suất I/O - một trong những nguyên nhân chính khiến performance trở nên kém.

## Một ****số thuật ngữ liên quan****

* **Non-relational**: relational - ràng buộc - thuật ngữ sử dụng chỉ đến các mối quan hệ giữa các bảng trong cơ sở dữ liệu quan hệ (RDBMs) sử dụng mô hình khóa gồm 2 loại khóa: khóa chính và khóa phụ (primary key + foreign key) để ràng buộc dữ liệu nhằm thể hiện tính nhất quán dữ liệu từ các bảng khác nhau.   
  Non-relational là khái niệm không sử dụng các ràng buộc dữ liệu cho nhất quán dữ liệu ở NoSQL database.



*So sánh cách thiết kế giữa NoSQL và RDBMS*

Nhìn vào hình trên ta thấy NoSQL có cách thiết kế lỏng lẻo, không ràng buộc chặc chẽ như RDBMS. Các mối liên kết giữa các Node trong NoSQL chỉ là liên kết ảo, NoSQL không nhìn thất mối liên kết gì ở đây cả. Tuy nhiên nhờ bỏ qua tính ràng buộc này đã giúp cho NoSQL có khả năng làm việc tốt với lượng dữ liệu lớn.

* **Distributed storage**: mô hình lưu trữ phân tán các file hoặc dữ liệu ra nhiều máy tính khác nhau trong mạng LAN hoặc Internet  dưới sự kiểm soát của phần mềm.
* **Eventual consistency (nhất quán cuối)**: tính nhất quán của dữ liệu không cần phải đảm bảo ngay tức khắc sau mỗi phép write. Một hệ thống phân tán chấp nhận những ảnh hưởng theo phương thức lan truyền và sau một khoảng thời gian (không phải ngay tức khắc), thay đổi sẽ đi đến mọi điểm trong hệ thống, tức là cuối cùng (eventually) dữ liệu trên hệ thống sẽ trở lại trạng thái nhất quán.
* **Vertical scalable (khả năng mở rộng chiều dọc)**:  Khi dữ liệu lớn về lượng,  phương pháp tăng cường khả năng lưu trữ và xử lý bằng việc cải tiến phần mềm và cải thiện phần cứng trên một máy tính đơn lẻ được gọi là khả năng mở rộng chiều dọc. Ví dụ việc tăng cường CPUs, cải thiện đĩa cứng, bộ nhớ trong một máy tính... cho RDBMs nằm trong phạm trù này. Khả năng mở rộng chiều dọc còn có một thuật ngữ khác scale up.
* **Horizontal scalable (khả năng mở rộng chiều ngang)**:
  + Khi dữ liệu lớn về lượng, phương pháp tăng cường khả năng lưu trữ và xử lý  là dùng nhiều máy tính phân tán. Phân tán dữ liệu được hỗ trợ bởi phần mềm tức cơ sở dữ liệu.
  + Trong khi giá thành phần cứng ngày càng giảm, tốc độ xử lý, bộ nhớ ngày càng tăng thì  horizontal scalable là một lựa chọn đúng đắn. Hàng trăm máy tính nhỏ được chập lại tạo thành một hệ thống tính toán mạnh hơn nhiều so với vi xử lý RISC truyền thống đơn lẻ. Mô hình này tiếp tục được hỗ trợ bởi các công nghệ kết nối Myrinet và InfiniBand. Từ đó chúng ta có thể quản lý, bảo trì từ xa, xây dựng batch procession (xử lý đồng loạt tập lệnh) tốt hơn. Do những đòi hỏi về tốc độ xử lý I/O cao, lượng cực lớn dữ liệu,...  scale horizontally sẽ thúc đẩy các công nghệ lưu trữ mới phát triển giống như object storage devices (OSD).

## Ưu nhược điểm của cơ sở dữ liệu NoSQL:

### Ưu điểm:

* **Nó là nguồn mở:** Các sản phẩm nguồn mở đưa ra cho những người phát triển với nhiều lợi  ích to lớn, trong đó việc sử dụng miễn phí là một lợi ích lớn. Những  lợi ích khác: phần mềm nguồn mở có xu hướng sẽ là tin cậy hơn, an ninh  hơn và nhanh hơn để triển khai so với các lựa chọn thay thế sở hữu độc quyền.Ví dụ như các hệ quản trị cơ sở dữ liệu (CSDL) NoSQL: Cassandra, CouchDB, Hbase, RavenDB, MongoDB và Redis.
* **Việc mở rộng phạm vi là mềm dẻo**: NoSQL thay thế câu thần chú cũ của các nhà quản trị CSDL về 'mở rộng  phạm vi' với một thứ mới: 'mở rộng ra ngoài'. Thay vì bổ sung thêm các  máy chủ lớn hơn để điều khiển nhiều tải dữ liệu hơn, thì CSDL NoSQL cho  phép một công ty phân tán tải qua nhiều máy chủ khi mà tải gia tăng.
* **Các CSDL NoSQL khác nhau cho những dự án khác nhau**:
  + MongoDB và Redis là những lựa chọn tốt cho việc lưu trữ các dữ liệu  thống kê ít được đọc mà lại được viết thường xuyên, như một số đếm truy  cập web chẳng hạn.
  + Hadoop, một CSDL dạng tự do, phân tán làm tốt công việc lưu trữ các dữ  liệu lớn như các con số thống kê thời tiết hoặc công việc phân tích  nghiệp vụ.
  + Memcache, một CSDL nhất thời chóng tàn, tuyệt vời trong lưu trữ các  phiên làm việc web, các khóa, và các con số thống kê ngắn hạn.
  + Cassandra và Riak (các lưu trữ dư thừa, tự động tạo bó cluster) làm tốt  trong các môi trường với các ứng dụng có tính sẵn sàng cao, khi thời  gian sống tối đa là sống còn.
* **NoSQL được các hãng lớn sử dụng**: Các công ty như Amazon, BBC, Facebook và Google dựa vào các CSDL NoSQL.
* **NoSQL phù hợp với công nghệ đám mây**: NoSQL và đám mây là một sự trùng khớp tự nhiên. Các máy chủ ngày nay là  không đắt và có thể dễ dàng mở rộng phạm vi được theo yêu cầu có sử dụng  một dịch vụ như là Amazon EC2. Giống như tất cả công nghệ đám mây, EC2  dựa vào ảo hóa. Liên kết yếu của ảo hóa là sự thực thi của I/O, với bộ  nhớ và CPU các các kết nối mạnh.
* Các CSDL NoSQL hầu hết sử dụng bộ nhớ qua đĩa như là vị trí ghi đầu tiên  - vì thế ngăn ngừa được sự thực thi không ổn định của I/O. Và vì NoSQL  lưu trữ dữ liệu thường thúc đẩy được tính mở rộng phạm vi theo chiều  ngang thông qua việc ngăn chia, chúng có khả năng tận dụng được việc  cung cấp mềm dẻo của đám mây.

### Nhược điểm:

* **Nguồn mở có thể có nghĩa là sự hỗ trợ không đồng đều cho các doanh nghiệp:** 
  + Trong khi các nhà cung cấp chủ chốt của RMBMs như Oracle, IBM hay Sybase  đưa ra sự hỗ trợ tốt nổi tiếng cho các khách hàng doanh nghiệp cỡ vừa,  thì các doanh nghiệp nhỏ hơn, thường là các nhà cung cấp nguồn mở mới  thành lập không thể mong đợi được cung cấp sự hỗ trợ có thể so sánh  được (ngoại trừ một nhóm các khách hàng blue chip).
  + Nhà cung cấp nguồn mở trung bình thiếu sự tiếp cận toàn cầu, các dịch vụ hỗ trợ và sự tin cậy của Oracle hay IBM.
* **Chưa đủ “chín” cho các doanh nghiệp**: Dù chúng đã được triển khai tại một số công ty lớn thì các CSDL NoSQL  vẫn đối mặt với một vấn đề về sự tin cậy chính với nhiều doanh nghiệp. Điểm sống còn của NoSQL là thiếu về độ “chín” muồi và các vấn đề về tính không ổn định, trong khi đó tính chín muồi, hỗ trợ đầy đủ chức năng và tính ổn định của các RDBMs được thiết lập đã từ lâu.
* **Những hạn chế về tri thức nghiệp vụ**: Có một vài câu hỏi xung quanh những khả năng về tri thức nghiệp vụ (BI)  của các CSDL NoSQL. Liệu các CSDL này có thể cung cấp dạng phân tích dữ  liệu lớn và mạnh mà các doanh nghiệp đã quen với các RDBMS? Cần bao  nhiêu sự tinh thông về lập trình cần có để tiến hành những truy vấn và  phân tích hiện đại?
  + Các câu trả lời là không tích cực. Các CSDL NoSQL không có nhiều sự đeo bám tới các công cụ BI thường được sử dụng, trong khi những yêu cầu và  phân tích hiện đại đơn giản nhất thì cũng liên quan khác nhiều tới sự tinh  thông về lập trình. Tuy vậy, các giải pháp là sẵn sàng. Quest Software,  ví dụ, đã tạo ra Toad cho các CSDL đám mây, mà nó phân phối các khả năng  truy vấn hiện đại tới một số CSDL NoSQL.
* **Thiếu sự tinh thông**: Tính rất mới mẻ của NoSQL có nghĩa là không có nhiều lập trình viên và  người quản trị mà biết công nghệ này - là khó khăn cho các công ty  tìm người với sự tinh thông phù hợp. Đối lại, thế giới của RDBMS có hàng  ngàn những người đủ tư cách.
* **Những vấn đề về tính tương thích**: Không giống như các CSDL quan hệ, các CSDL NoSQL chia sẻ ít theo cách  thức của các tiêu chuẩn. Mỗi CSDL NoSQL có các giao diện lập trình ứng  dụng API riêng của mình, các giao diện truy vấn độc nhất vô nhị, và  những sự riêng biệt. Sự thiếu hụt các tiêu chuẩn có nghĩa là nó không có  khả năng để chuyển một cách đơn giản từ một nhà cung cấp này sang một  nhà cung cấp khác nếu bạn không hài lòng với dịch vụ.

## Cách triển khai một ứng dụng NoSQL

Trong phạm vi của luận văn này và sự phong phú cách lưu trữ của cơ sở dữ liệu NoSQL, chúng tôi chỉ tập trung vào loại phổ biến nhất trong cơ sở dữ liệu NoSQL đó là loại Document Store. Do đó trong mục này, chúng tôi sẽ trình bày cách triển khai một ứng dụng sử dụng cơ sở dữ liệu NoSQL loại Document Store.

Để hiểu rõ các loại này, vui lòng xem “Chương 3: Tìm hiểu các giải pháp cơ sở dữ liệu NoSQL”.

### Xác định NoSQL có phù hợp

Khi làm việc với một lượng lớn dữ liệu, bạn hãy nghĩ đến NoSQL. NoSQL rất thích hợp để làm việc với dữ liệu lớn bằng cách loại bỏ các ràng buộc toàn vẹn dữ liệu, cách thiết kế mô hình phi chuẩn hoá, cách sử dụng index…. Đã giúp NoSQL trở nên mạnh mẽ để làm việc với lượng lớn dữ liệu. Tuy nhiên, có một số tính chất sau đây cần lưu ý khi lựa chọn cơ sở dữ liệu NoSQL.

Như đã đề cập trong mục 2.5 Một số thuật ngữ liên quan, tính nhất quán cuối (Eventual consistency)cần phải được ứng dụng chấp nhận. Có nghĩa là ứng dụng không yêu cầu ràng buộc dữ liệu, không yêu cầu dữ liệu phải cập nhập chính xác ngay tức thì. Một số ứng dụng phù hợp như các trang mạng xã hội, các ứng dụng ghi log tự động… Các ứng dụng loại này chấp nhập dữ liệu cũ trong một khoảng thời gian ngắn trước khi được cập nhập mới. Đổi lại chúng ta đạt được những tiêu chuẩn cao về khả năng mở rộng và hiệu quả về chi phí, trong khi phục vụ liên tục hàng triệu khách hàng từ khắp nơi trên trái đất. Đặt biệt chúng ta đạt được một hiệu suất hoạt động cao hơn gấp nhiều lần nhờ vào việc loại bỏ các yêu cầu nhất quán dữ liệu.

Các ứng dụng không phù hợp với cơ sở dữ liệu NoSQL là các ứng dụng yêu cầu tính nhất quán dữ liệu cao. Tính nhất quán dữ liệu được xem như tính sống còn của ứng dụng. Ví dụ như các ứng dụng tài chính, ngân hàng… với các con số luôn được cập nhập và cần được cập nhập tức thì. Sự chậm trễ có thể phải trả giá rất đắt. Bởi thế nếu các ứng dụng của bạn thuộc loại này thì hãy lựa chọn cơ sở dữ liệu RDBMs với mô hình quan hệ truyền thống.

Các yêu cầu phân tích hiện đại (BI) cũng không phù hợp với cơ sở dữ liệu NoSQL này. Bởi vì NoSQL hổ trợ rất ít các câu truy vấn. Tất cả đều phụ thuộc vào sự tinh thông lập trình. Như vậy, với một yêu cầu phân tích đơn giản thì cũng cần đến lập trình trong đó. Trong khi với cơ sở dữ liệu RDBMs sử dụng ngôn ngữ SQL để truy vấn, SQL giúp chúng ta rất nhiều việc trong truy vấn, phân tích.

### Thiết kế cấu trúc dữ liệu dạng document

NoSQL 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.

NoSQL 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 document database. Hãy xem xét ví dụ sau đây để so sánh 2 cách thiết kế: thiết kế chuẩn hoá và thiết kế document:

Ví dụ yêu cầu quản lý thông tin sản phẩm (Product). Các thông tin của một sản phẩm gồm có: ID, giá, mô tả sản phẩm.

* + Đối với sản phẩm sách có thêm thông tin: tác giả, tiêu đề, ngày xuất bản.
  + Đối với sản phẩm Album nhạc có thêm thông tin: nhạc sĩ, tên Album. Trong mỗi Album có nhiều bài hát, mỗi bài hát có tên tên bài hát.
  + Đối với sản phẩm quần Jean có thêm thông tin: Model, chiều dài,chiều rộng.



*Ví dụ về thiết kế dữ liệu chuẩn hoá và document của NoSQL*

Với thiết kế chuẩn hoá, các table quan hệ khoá ngoại với nhau tạo nên tính nhất quán dữ liệu. Nhưng với cách thiết kế document, chúng ta gom tất cả vào một document và không chia ra nhiều table. Nên khi cần truy xuất dữ liệu, chúng ta chỉ cần một vài truy vấn đã lấy được tất cả dữ liệu cần thiết mà không cần dùng đến các khoá ngoại rườm rà.

Tóm lại, tư tưởng thiết kế ở đây là đi ngược lại với thiết kế chuẩn hoá, mục tiêu sao cho hạn chế các phép “join” rườm rà. Ở đây chúng ta có thể chấp nhập dữ liệu dư thừa và không thống nhất trong 1 khoảng thời gian và sau đó sẽ được cập nhập lại. Bù lại ta nhận được một hiệu suất hoạt động mạnh mẽ với lượng lớn dữ liệu.

Vấn đề đặt ra khi ta cần cập nhập dữ liệu. Như ví dụ sau đây, tên của “Customer” cần được cập nhập. Đối với thiết kế chuẩn hoá, ta chỉ cần cập nhập ở 1 nơi là table Customer. Nhưng đối với thiết kế document thì khác, tên của Customer đặt ở nhiều nơi: trong object Customer và trong các object Order. Đến đây thì không có một quy tắt nào hết. Việc cập nhập lại tên của Customer là phụ thuộc vào chương trình. Khi xây dựng chương trình, ta phân tích xem tên của Customer có cần được cập nhập ở tất cả các nơi hay chỉ cần ở 1 số nơi. Từ đó ta sẽ viết code cho việc cập nhập này. Tất cả đều do phân tích cho từng chương trình sao cho hiệu suất hoạt động tốt nhất và nghiệp vụ vẫn đúng.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Normalization** | | |  |  | **Document** | | |
|  |  |  |  |  |  |  |  |
| **Customer** |  | **Product** |  |  | **Customer** |  | **Product** |
| Id |  | Id |  |  | Id |  | Id |
| Name |  | Name |  |  | Name |  | Name |
| Phone |  | Price |  |  | Phone |  | Price |
| Address |  |  |  |  | Address |  |  |
|  |  |  |  |  |  |  |  |
| **Order** |  |  |  |  | **Order** |  |  |
| Id |  |  |  |  | Id |  |  |
| CustomerId |  |  |  |  | CustomerId |  |  |
| OrderDate |  |  |  |  | CustomerName |  |  |
|  |  |  |  |  | OrderDate |  |  |
| **OrderDetail** |  |  |  |  | ListProducts: |  |  |
| Id |  |  |  |  |  | ProductId |  |
| OrderId |  |  |  |  |  | ProductName |  |
| ProductId |  |  |  |  |  | Price |  |
| Price |  |  |  |  |  | Quantity |  |
| Quantity |  |  |  |  |  | … |  |
|  |  |  |  |  |  |  |  |

### Ví dụ so sánh cách thiết kế mô hình dữ liệu trong RDMBs và Document Database:

Ta có mô hình dữ liệu cho ứng dụng blog đơn giản trong cơ sở dữ liệu quan hệ như sau:



Sẽ là sai lầm lớn nếu chúng ta tiếp cận với document database khi cố gắng đi theo mô hình quan hệ và áp dụng nó vào document database. Đoạn code bên dưới là dữ liệu chúng ta muốn lưu trữ:

|  |
| --- |
| *var user = new User("ayende");*  *var blog = new Blog("Ayende @ Rahien", user) { Tags = {".NET", "Architecture", "Databases" } };*  *var categoryRaven = new Category("Raven");*  *var categoryNoSQL = new Category("NoSQL");*  *var post = new Post(blog, "RavenDB", "... content ...")*  *{*  *Categories = { categoryRaven, categoryNoSQL },*  *Tags = {"RavenDB", "Announcements" }*  *};*  *var comment = new Comment(post, "Great news");*  *PersistAll(user, blog, categoryRaven, categoryNoSQL, post, comment);* |

Chúng ta sẽ sử dụng code để biểu diễn dữ liệu mà không ràng buộc dữ liệu vào định dạng cụ thể nào.

Lưu dữ liệu không đúng cách là lưu trữ dữ liệu của mỗi đối tượng như một document riêng biệt, tương tự như cách chúng ta lưu trữ từng đối tượng thành một hàng riêng trong cơ sở dữ liệu quan hệ. Lưu trữ không đúng cách trông sẽ như bên dưới:

|  |  |  |
| --- | --- | --- |
| |  | | --- | | *// users/ayende*  *{*  *"type": "user",*  *"name": "ayende"*  *}*  *// tags/1*  *{*  *"name": ".NET"*  *}*  *// tags/2*  *{*  *"name": "Architecture"*  *}*  *// tags/3*  *{*  *"name": "Databases"*  *}*  *// tags/4*  *{*  *"name": "RavenDB"*  *}*  *// tags/5*  *{*  *"name": "Announcements"*  *}* | | *// categories/1*  *{*  *"name": "Raven"*  *}*  *// categories/2*  *{*  *"name" : "NoSQL"*  *}*  *// blogs/1*  *{*  *"type": "blog",*  *"users": ["users/ayende"],*  *"name": "Ayende @ Rahien",*  *"tags": ["tags/1", "tags/2", "tags/3"]*  *}*  *// posts/1*  *{*  *"blog": "blogs/1",*  *"title": "RavenDB",*  *"content": "... content ...",*  *"categories": ["categories/1", "categories/2"]*  *"tags" : ["tags/4", "tags/5"]*  *}*  *// comments/1*  *{*  *"post": "posts/1",*  *"content": "Great News"*  *}* | |

Lưu ý là không nên mô hình một document trong document database theo cách này.

Lý do chính là: điều này sẽ sai vì một document database không thật sự hỗ trợ việc joins, unions hay bất cứ thứ gì làm cho mô hình làm việc hiệu quả trong mô hình cơ sở dữ liệu.

Sau đây là một số yêu cầu khi hiển thị dữ liệu trong ứng dụng blog trên là:

* + Trang chính: hiển thị danh sách các blogs
  + Trang chính: hiển thị danh sách các bài viết mới nhất
  + Trang chính: hiện thị danh sách các bình luận mới nhất
  + Trang chính: hiển thị các tags của bài viết
  + Trang chính: hiển thị danh mục
  + Trang viết bài: hiển thị bài viết và tất cả bình luận
  + Trang viết bài: thêm bình luận cho bài viết
  + Trang viết bài: hiển thị tất cả bài viết theo tag
  + Tràn danh mục: hiển thị danh sách bài viết theo danh mục

Bây giờ chúng ta sẽ đi vào từng yêu cầu và thực hiện truy vấn trên cả SQL và tương ứng với mỗi truy vấn SQL đó chúng ta có những truy vấn của document database (lưu ý là: những truy vấn theo dạng document database chỉ là cách làm tương tự như SQL với truy vấn LINQ, còn những truy vấn thực sự của document database chúng ta sẽ nói ở phần tiếp theo)

* Trang chính: hiển thị danh sách các blogs
  + Sử dụng SQL:

|  |
| --- |
| *select \* from blogs* |

* + Sử dụng DocDB, sử dụng index có sẵn để truy vấn dữ liệu dựa vào kiểu dữ liệu:

|  |
| --- |
| *docDb.Query<Blog>("DocumentsByType", query:"type:blog");* |

* Trang chính: hiển thị danh sách các bài viết mới nhất
  + Sử dụng SQL:

|  |
| --- |
| *select \* from posts order by PostedAt desc* |

* + Sử dụng DocDB, sử dụng index do chúng ta định nghĩa, cho phép sắp xếp trong nó(không khó khăn lắm):
    - Index fuction:

|  |
| --- |
| *from doc in docs*  *where doc.type == "post"*  *select new {doc.posted\_at}* |

* + - Truy vấn dữ liệu sử dụng:

|  |
| --- |
| *docDb.Query<Post>("Posts", orderBy:"-posted\_at");* |

* Trang chính: hiện thị danh sách các bình luận mới nhất: tương tự bài viết mới nhất
* Trang chính: hiển thị tag
  + Sử dụng SQL:

|  |
| --- |
| *select Name, COUNT(\*) as TagCount from tags*  *where ItemType = 'Posts'*  *group by Name* |

* + Sử dụng DocDB, chúng ta cần viết một map/reduce index:
    - Map/ reduce index:

|  |
| --- |
| *from doc in docs*  *where doc.type == "posts"*  *from tag in doc.tags*  *select new { tag, count = 1 }*  *from result in results*  *group result by result.tag into g*  *select new { tag = g.Key, count = g.Sum(x=>x.count) }* |

* + - Khi đã có map/reduce index chúng ta sử dụng nó để lấy dữ liệu:

|  |
| --- |
| *var tagCloud = new TagCloud();*  *var tagIds = docDb.Query<TagAndCount>("TagsCloud", orderBy:"+count");*  *foreach(var tagId in tagIds)*  *{*  *var tag = docDb.Get<Tag>(tagId.Tag);*  *tagCloud.Add(tag.Name, tagId.Count);*  *}* |

* Trang chính: hiển thị danh mục: tương tự blogs
* Trang viết bài: hiển thị bài viết và tất cả bình luận
  + Sử dụng SQL:

|  |
| --- |
| *select \* from Posts where Id = 1*  *select \* from Comments where PostId = 1* |

* + Để hiệu quả hơn chúng ta dùng join:

|  |
| --- |
| *select \* from Posts*  *join Comments*  *on Posts.Id = Comments.Id*  *where Posts.Id = 1* |

* + Với DocDB, chúng ta có thể làm như sau

|  |
| --- |
| *var post = docDb.Get<Post>(1);*  *var comments = docDb.Query<Comment>("CommentsByPost", query:"post\_id:1", orderBy:"+posted\_at");* |

Với index CommentsByPost được định nghĩa như sau

|  |
| --- |
| *from doc in docs*  *where doc.type == "comment"*  *select new{doc.post\_id, doc.posted\_at}* |

* + Lưu ý rằng chúng ta phải thực hiện 2 lời gọi ở đây bởi vì document database không có khái niệm join
* Trang viết bài: thêm bình luận
  + Sử dụng SQL:

|  |
| --- |
| *insert into comments (PostId, ... )*  *values(1, ...)* |

* + Sử dụng DocDB:

|  |
| --- |
| *docDb.Store(new Comment{ PostId = 1, ... });*  *docDb.SaveChanges();* |

* Tag page: hiển thị tất cả bài viết theo tag
  + Sử dụng SQL thì hơi phức tạp một chút bởi vì tags có liên kết với blogs và posts. Chúng ta sẽ làm như sau:

|  |
| --- |
| *select \* from Posts*  *where Id in (*  *select ItemId from tags*  *where ItemType = 'Posts' and TagId = 1*  *)* |

* + Sử dụng DocDB:

|  |
| --- |
| *var posts = docDb.Query<Post>("PostsByTag", query:"tag:tags/1");* |

Với index được định nghĩa:

|  |
| --- |
| *from doc in docs*  *where doc.type == "post"*  *from tag in doc.tags*  *select new {tag}* |

* Trang danh mục: hiển thị tất cả các bài viết theo danh mục: tương tự tags

Nhận xét: bằng cách sao chép mô hình dữ liệu cơ sở quan hệ, chúng ta đã tạo ra một số thứ mà chúng ta đã từng làm với cơ sở dữ liệu quan hệ. Và bây giờ chúng ta gặp phải vấn đề là document database không thể làm được những điều như cơ sở dữ liệu quan hệ có thể làm. Những gì chúng ta làm ở trên thật uổng phí. Chúng ta đạt được một số lợi ích nhỏ khi sử dụng schemaless (phi lược đồ), nhưng điều đó không thực sự có lợi so với những nỗ lực của chúng ta phải làm bằng cách cố gắng có được cơ sở dữ liệu quan hệ trên một cơ sở dữ liệu phi quan hệ.

Bây giờ chúng ta sẽ thiết kế mô hình dữ liệu sử dụng document database đúng cách.

Khi thiết kế mô hình dữ liệu sử dụng document database chúng ta sẽ nghĩ đến Aggregates(kiểu tập hợp). Những thực thể ở mô hình trên có phải là những Aggregates không? User, Blog và Post là những Aggregate vì chúng có thể tồn tại một cách độc lập. Những thực thể còn lại không thể tồn tại riêng biệt được. Tags là Value Objects (những đối tượng chứa giá trị), Categories dùng để liệt kê dữ liệu, Comments thì không thật sự có ý nghĩa nếu tồn tại bên ngoài bài viết của nó (Post). Định dạng được lựa chọn đó là:

|  |  |  |
| --- | --- | --- |
| |  | | --- | | // users/ayende  {  "type": "user",  "name": "ayende"  }  // blogs/1  {  "type": "blog",  "users": ["users/ayende"],  "name": "Ayende @ Rahien",  "tags": [".NET", "Architecture", "Databases"]  } | | // posts/1  {  "blog": "blogs/1",  "title": "RavenDB",  "content": "... content ...",  "categories": ["Raven", "NoSQL"]  "tags" : ["RavenDB", "Announcements"],  "comments": [  { "content": "Great News" }  ]  } | |

Điều này làm cho mô hình trở nên nhỏ hơn và làm nhiều thứ như lấy dữ liệu của bài viết và những bình luận của nó ít tốn chi phí. Chúng ta xem lại một số yêu cầu của ứng dụng vơi mô hình mới này.

* Trang chính: hiển thị danh sách các blogs

|  |
| --- |
| *var blogs = docDb.Query<Blog>("DocumentsByType", "type:blog");* |

* Trang chính: hiển thị danh sách các bài viết mới nhất
  + - Index fuction:

|  |
| --- |
| *from doc in docs*  *where doc.type == "post"*  *select new {doc.posted\_at}* |

* + - Truy vấn dữ liệu sử dụng:

|  |
| --- |
| *docDb.Query<Post>("Posts", orderBy:"-posted\_at");* |

* Trang chính: hiển thị danh sách các bình luận mới nhất
  + Đây là một điều thú vị vì bây giờ chúng ta không còn khái niệm Comments như một thực thể riêng biệt. Chúng ta định nghĩa index để rút trích thông tin từ bài viết:

|  |
| --- |
| *from doc in docs*  *where doc.type == "post"*  *from comment in doc.comments*  *select new {comment.posted\_at, comment.content }* |

* + Và truy vấn như sau:

|  |
| --- |
| *var recentComments = docDb.Query<Comment>("CommentsByTime", orderBy:"-posted\_at");* |

* Trang chính: hiển thị các tag cho những bài viết
  + Đây cũng là một điều thú vị khác, chúng ta cần một tập hợp các Tags. Nó được thực hiện bởi thao tác Map/Reduce. Nghe thì có vẻ khủng khiếp nhưng thực sự thì dễ dàng bằng cách sử dụng 2 câu lệnh LINQ:

|  |
| --- |
| *from doc in docs*  *where doc.type == "post"*  *select tag in doc.tags*  *select new { tag, count = 1}*  *from result in results*  *group result by result.tag into g*  *select new {tag = g.Key, count = g.Sum(x=>x.count) }* |

* + Và chúng ta dùng nó để truy vấn dữ liệu:

|  |
| --- |
| *var tagCloud = docDb.Query<TagAndCount>("TagCloud");* |

* Trang chính: hiển thị danh mục
  + Tương tự như Tags, chúng ta rút trích thông tin categories từ Posts, và lấy số lượng bài viết trong mỗi categories:

|  |
| --- |
| *from doc in docs*  *where doc.type == "post"*  *select category in doc.categories*  *select new { category, count = 1}*  *from result in results*  *group result by result.category into g*  *select new {category= g.Key, count = g.Sum(x=>x.count) }* |

* + Và truy vấn dữ liệu:

|  |
| --- |
| *var categories = docDb.Query<CategoryAndCount>("CategoriesCount");* |

* Trang viết bài: hiển thị bài viết và tất cả các bình luận
  + Thật dễ dàng để quản lý đối tượng Post, tất cả những gì chúng ta cần là lấy post document, và chúng ta lấy mọi thứ chúng ta cần để hiển thị trên trang post:

|  |
| --- |
| *var post = docDb.Get<Post>("posts/1");* |

* Trang viết bài: thêm bình luận cho bài viết
  + Nên nhớ rằng, comment không thể tồn tại bên ngoài Post được, vì vậy để thêm comment thì chúng ta sẽ lấy Post, thêm Comment và lưu lại:

|  |
| --- |
| *var post = docDb.Get<Post>("posts/1");*  *post.Comments.Add(new Comment(...));*  *docDb.SaveChanges();* |

* Trang tag: hiển thị tất cả bài viết theo tag
  + Index

|  |
| --- |
| *from doc in docs*  *where doc.type == "post"*  *from tag in doc.tags*  *select new { tag }* |

* + Truy vấn như sau

|  |
| --- |
| *var posts = docDb.Query<Post>("PostsByTag", "tag:Raven");* |

* Trang danh mục: hiển thị tất cả bài viết theo danh mục, tương tự hiển thị tất cả bài viết theo tag
* Một số lưu ý:
  + Để có kỹ thuật thiết kế mô hình tốt thì nghĩ tới Aggregates trong DDD(Domain Driven Design), một Aggregate và tất cả các liên quan tới nó đều được chứa trong một document duy nhất.
  + Nếu chúng ta muốn truy vấn thông tin một thực thể có trong Aggregate thì chúng ta sử dụng index để rút trích thông tin.
  + Chúng ta có thể tạo ra các thực thể ảo (“virtual entities”) như là Tags hay Categories, những thực thể không thực sự tồn tại, nhưng được tạo ra trong index khi chúng ta định nghĩa. Những thực thể ảo hữu dụng khi dùng trong tìm kiếm/tổ chức một Aggregate.
  + Lược đồ không quan trọng bởi vì tìm kiếm dựa trên key những well-known indexs , trả về những định dạng dữ liệu mà chúng ta mong đợi.

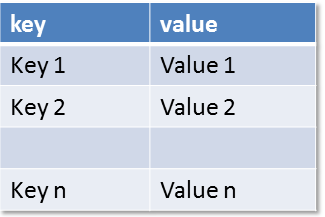
# CHƯƠNG 3: CÁC GIẢI PHÁP CƠ SỞ DỮ LIỆU NOSQL

Thông thường, cơ sở dữ liệu NoSQL được phân loại theo cách mà nó lưu trữ dữ liệu và thuộc các loại sau: key-value store, wide column store/ column families với triển khai nổi tiếng là BigTable của Google, document database và graph databases. Với sự gia tăng của các trang web thời gian thực, nơi mà cần được cung cấp một lượng lớn thông tin dữ liệu nhiều hơn hoặc ít hơn theo cấu trúc ngang (mở rộng theo chiều ngang). Như vậy, cơ sở dữ liệu NoSQL chú trọng vào việc tối ưu hóa cho thao tác lấy dữ liệu lên, các thao tác dữ liệu nối tiếp nhau và thường cung cấp ít chức năng vượt ra ngoài lưu trữ dữ liệu (ví dụ như key-value store). Các hoạt động giảm thời gian một cách linh hoạt so với hệ thống SQL đầy đủ và được bù đắp bởi khả năng mở rộng và hiệu suất cao cho các mô hình dữ liệu cố định.

Triển khai NoSQL có thể phân loại theo cách nó được hiện thực. Có 4 loại cơ sở dữ liệu NoSQL chính:

## Key-Value Store

Cơ sở dữ liệu NoSQL đơn giản nhất chính là Key/Value stores.



Nó đơn giản nhất là vì những API của nó đơn giản, những triển khai thực tế của NoSQL thường rất phức tạp. Hầu hết Key/Value stores thường có những API sau:

|  |
| --- |
| *void Put(string key, byte[] data);*  *byte[] Get(string key);*  *void Remove(string key);* |

Có rất nhiều biến thể nhưng Key/Value store là cơ sở cho tất cả những biến thể đó. Một Key/Value store cho phép chúng ta lưu trữ giá trị bởi khóa (key-value). Giá trị được lưu dưới dạng BLOB (Binary large object). Đơn giản là lưu trữ dữ liệu mà không quan tâm đến nội dung lưu trữ. Nói cách khác, chúng ta lưu trữ dữ liệu mà không cần xác định lược đồ, nhưng phía client thì có định nghĩa mang tính tham khảo để biết dữ liệu thực sự như thế nào. Lợi ích của phương pháp này là rất đơn giản để xây dựng một key/value store và nó rất dễ dàng mở rộng. Một key/value store có hiệu suất rất tốt bởi vì mô hình truy cập dữ liệu trong key/value store được chú trọng rất nhiều vào việc tối ưu hóa. Nói chung, hầu hết các thao tác key/value được thực hiện sử dụng O(1) bất kể có bao nhiêu máy dùng để lưu trữ dữ liệu và dữ liệu có độ lớn như thế nào.

Trường hợp sử dụng: Key-value store rất hữu ích khi chúng ta cần truy cập dữ liệu theo khóa. Người dùng có những dữ liệu liên quan, chẳng hạn như phiên giao dịch hoặc là ý tưởng thông tin giỏ hàng bởi vì chúng ta luôn biết id của người dùng. Một trường hợp sử dụng phổ biến nữa là để lưu trữ dữ liệu tính toán trước dựa trên khóa chính. Ví dụ, chúng ta muốn lưu trữ tất cả thông tin về một sản phẩm (bao gồm cả những sản phẩm liên quan, đánh giá…) trong một key-value store dựa trên Product SKU. Điều đó cho phép chúng ta truy vấn tất cả thông tin về một sản phẩm dựa trên O(1) (hằng số). Bởi vì những truy vấn dựa trên khóa thì miễn phí, vì thế cấu trúc dữ liệu được truy cập cùng khóa chúng ta sẽ được hiệu suất đáng kể bằng cách cấu trúc ứng dụng phù hợp theo yêu cầu đó. Nó cũng chỉ ra rằng là có khá nhiều thứ chúng ta có thể làm chỉ với key-value store. Giỏ mua hàng của Amazon chạy trên key value store (Amazon Dynamo). Vì thế có thể thấy rằng key-value store có khả năng mở rộng cao. Amazon Dynamo Paper là một ví dụ tốt nhất về kiểu dữ liệu key-value store. Rhino DHT có khả năng mở rộng, chuyển đổi dự phòng, không cấu hình, là dạng key-value store trên nền tảng .Net.

## Wide Column Store / Column Families

Column family databases được biết đến nhiều nhất thông qua sự triển khai **BigTable** của Google. Nhìn bên ngoài vào nó giống với cơ sở dữ liệu quan hệ nhưng thực sự thì có sự khác biệt rất lớn từ bên trong. Một trong những khác biệt đó chính là việc lưu trữ dữ liệu theo dòng (trong cơ sở dữ liệu quan hệ) so với việc lưu trữ dữ liệu theo cột (trong column family databases). Nhưng sự khác biệt lớn là ở chính khái niệm của nó. Chúng ta không thể áp dụng cùng một giải pháp mà chúng ta sử dụng trong cơ sở dữ liệu quan hệ vào trong cơ sở dữ liệu column family. Đó là bởi vì cơ sở dữ liệu cột (column family database) phi quan hệ. Các khái niệm sau đây rất quan trọng để hiểu được column family database làm việc như thế nào:

* Column family (cột quan hệ)
* Super column (siêu cột)
* Column (cột)

Column và super column trong column family database dùng thay thế nhau, có nghĩa là chúng sẽ là 0 byte nếu chúng không có chứa dữ liệu. Không giống như một bảng, thứ duy nhất chúng ta cần xác định trong column family database tên cột và các tùy chọn chính (không có lược đồ cố định).

Ý tưởng cơ bản:

* Column families: Một column family là cách thức dữ liệu được lưu trữ trên đĩa. Tất cả dữ liệu trong một cột sẽ được lưu trên cùng một file. Một column family có thể chứa super column hoặc column.



* Super column: Một super column có thể được dùng như một dictionary(kiểu từ điển). Nó là một column có thể chứa những column khác (mà không phải là super column).



* Column: Một column là một bộ gồm tên, giá trị và dấu thời gian (thông thường chỉ quan tâm tới key-value).

Hiểu được lược đồ được thiết kế như thế nào rất quan trọng. Nếu chúng ta thiết kế lược đồ không đúng, chúng ta không thể lấy được dữ liệu. Column family database cung cấp hai cách truy vấn: key hoặc là key range. Điều này có nghĩa là, vì CFDB có thể phân tán nên khóa xác định vị trí vật lý thực sự mà dữ liệu được lưu trữ. Dữ liệu được lưu trữ dựa trên sự sắp xếp của column family và chúng ta không có cách nào để thay đổi sự sắp xếp này (ngoại trừ việc sắp xếp tăng dần hay giảm dần).

Không giống như trong cơ sở dữ liệu quan hệ, thứ tự sắp xếp không bị ảnh hưởng bởi giá trị cột, nhưng lại bị ảnh hưởng bởi tên cột.

Giả sử trong column family Users, một dòng với khóa chính là ”@ayende”, một cột name với giá trị “Ayende Rahien” và một cột location với giá trị “Israel”. CFDB sẽ sắp xếp vật lý trong User column family file như sau:

|  |
| --- |
| *@ayende/location = "Israel"*  *@ayende/name = "Ayende Rahien"* |

Bởi vì location đứng trước name (theo kiểu sắp xếp từ điển) nên cột location sẽ được lưu trước cột name. Tương tự cho super column, ví dụ cột Friends, “@ayende” có 2 friend thì trong file Friends column family được lưu trữ vật lý như sau:

|  |
| --- |
| *@ayende/friends/arava= 945*  *@ayende/friends/rose = 14* |

Điều này rất quan trọng để hiểu được cách làm việc của CFDB. Giả sử chúng ta có mô hình twitter cần lưu trữ: users và tweets. Chúng ta định nghĩa 3 cột:

* User: sắp xếp theo UTF8
* Tweets: sắp xếp theo Sequential Guid
* UsersTweets: super column, sắp xếp theo Sequential Guid

Giả sử chúng ta muốn tạo một User:

|  |
| --- |
| *cfdb.Users.Insert(key: "@ayende", name: "Ayende Rahien", location: "Israel", profession: "Wizard");* |

Chúng ta có thể xem hình bên dưới để biết được một dòng trông như thế nào, nó không giống với một dòng trong cơ sở dữ liệu quan hệ:



Biểu diễn một dòng trong Column family database

Giờ chúng ta sẽ tạo một tweet:

|  |
| --- |
| *var firstTweetKey = "Tweets/" + SequentialGuid.Create();*  *cfdb.Tweets.Insert(key: firstTweetKey, application: "TweekDeck", text: "Err, is this on?", private: true);*  *var secondTweetKey = "Tweets/" + SequentialGuid.Create();*  *cfdb.Tweets.Insert(key: secondTweetKey, app: "Twhirl", version: "1.2", text: "Well, I guess this is my mandatory hello world”, public: true, version: 1.2);* |



Hình 1.3: Biểu diễn 2 tweet trong Column family database

Giá trị được hiển thị trong hình 1.3. Một vài lưu ý là:

* Thực sự giá trị của khóa không quan trọng, nhưng nó là những chuỗi liên tiếp cho phép chúng ta sắp xếp sau này,
* Cả hai dòng chứa dữ liệu khác nhau bởi vì chúng ta không có lược đồ cố định.
* Chúng ta không có cách nào liên kết giữa một user và tweet

Trong cơ sở dữ liệu quan hệ, chúng ta sẽ định nghĩa một cột là UserId trong Tweet cho phép chúng ta liên kết với bảng User. Hơn nữa, cơ sở dữ liệu quan hệ còn cho phép chúng ta truy vấn tweets theo UserId. CFDB không làm được điều này, chúng ta không thể truy vấn theo dữ liệu cột. Thay vì thế, điều duy nhất CFDB cho phép chúng ta là truy vấn theo khóa. Chúng ta định nghĩa một index thứ hai, nơi mà cột UsersTweets xuất hiện:

|  |
| --- |
| *cfdb.UsersTweets.Insert(key: "@ayende", timeline: { SequentialGuid.Create(): firstTweetKey } );*  *cfdb.UsersTweets.Insert(key: "@ayende", timeline: { SequentialGuid.Create(): secondTweetKey } );* |

Hình 1.4 cho thấy dữ liệu trong database như thế nào. Chúng ta thêm dữ liệu vào cột UsersTweets một dòng với khóa là “@ayende”, super column timeline với 2 cột – tên mỗi cột là sequential guid cho phép sắp xếp.



Hình 1.4: biểu diễn index thứ hai, liên kết users và tweets trong Column Family database.

Lưu ý: Câu hỏi đặt ra là chúng ta có thể tạo ra super column trong cột User để lưu giữ quan hệ. Câu trả lời là: chúng ta có thể làm điều đó, ngoại trừ một column family có thể chứa hoặc column hoặc super column, không thể chứa cả hai.

Để lấy những tweets của một user, chúng ta cần thực thi:

|  |
| --- |
| var tweetIds =cfdb.UsersTweets.Get("@ayende")  .FetchSuperColumnValues("timeline")  .Take(25)  .OrderByDescending()  .Select(x=>x.Value);  var tweets = cfdb.Tweets.Get(tweetIds); |

Lưu ý: truy vấn này không phải là API cho .NET, chỉ là ví dụ cho dễ hiểu không phải là API thực sự.

Về cơ bản chúng ta thực hiện 2 truy vấn:

* Truy vấn thứ nhất vào cột UsersTweets, yêu cầu cột và giá trị trong timeline super column với khóa là “@ayende”
* Truy vấn thứ hai là truy vấn vào cột Tweets để lấy giá trị thực sự của Tweet.

Kiểu này chúng ta thường thấy trong NoSQL. Nó được gọi là ”secondary index”, một cách truy cập dữ liệu nhanh chóng theo khóa dựa trên một giá trị khác entity/row/document. Đây là một ví dụ về truy vấn Tweets theo User trên dữ liệu chúng ta có. Nếu không tạo ra secondary index thì chúng ta không có cách nào trả lời cho câu hỏi: “cho xem 25 tweets mới nhất của ayende?”

Bởi vì dữ liệu được sắp xếp theo tên cột và giảm dần, chúng ta có thể lấy được 25 tweets mới nhất của ayende. Và sẽ như thế nào nếu ta muốn truy vấn 25 tweets mới nhất (không theo một user nào)? Rất đơn giản, chỉ cần truy vấn vào cột Tweets sắp xếp theo khóa giảm dần.

**Tại sao column family database lại bị giới hạn?**

CFDB khó khăn trong việc nghiên cứu lúc đầu vì nó nhìn bên ngoài thì giống cơ sở dữ liệu quan hệ mà lại bị hạn chế rất nhiều. Không có “join”, không có khả năng truy vấn thực sự(ngoại trừ truy vấn theo khóa chính), không có sự phong phú sự hỗ trợ như cơ sở dữ liệu quan hệ làm được. SQLite hay Access đem lại nhiều lợi ích hơn CFDB. Tại sao CFDB lại hạn chế như vậy?

Câu trả lời khá đơn giản. CFDB được thiết kế để chạy trên một số lượng lớn các máy, và lưu trữ một lượng dữ liệu cực lớn. Chúng ta không thể lưu trữ một lượng lớn dữ liệu trong cơ sở dữ liệu quan hệ thậm chí là nhiều máy như Oracle RAC sẽ nhanh chóng bị sụp đổ hoặc là chết rất nhanh về kích thước của dữ liệu và những truy vấn đó được CFDB xử lý một cách dễ dàng. Nhớ rằng CFDB loại bỏ các khái niệm trừu tượng, những thứ làm cho nó cứng nhắt khi chạy trên một cụm máy.

Lý do mà CFDB không hỗ trợ join là join yêu cầu chúng ta quét toàn bộ tập hợp dữ liệu. Điều đó yêu cầu phải có một nơi có cái nhìn tổng quát về toàn bộ cơ sở dữ liệu(kết quả trong nút cổ chai và điểm duy nhất bị thất bại) hoặc thực hiện một truy vấn thực sự trên tất cả các máy có trong cụm. CFDB không cung cấp cách thức để truy vấn theo cột hay giá trị bởi vì nó sẽ yêu cầu một index trên toàn bộ tập hợp dữ liệu (hay chỉ là một cột duy nhất). Và một lần nữa không thực tế, không thể chạy truy vấn trên tất cả các máy. Bằng cách giới hạn truy vấn theo khóa, CFDB đảm bảo rằng nó biết chính xác về node mà truy vấn có thể thực hiện trên đó. Có nghĩa là mỗi truy vấn được chạy trên một tập nhỏ dữ liệu và cho chi phí rẻ hơn nhiều.

Nhiều hạn chế, khó sử dụng nhưng CFDB lại có khả năng mở rộng cao. Đây là điều chúng ta cần quan tâm tới.

## Document Store

### Giới thiệu

Document store thực chất là các document-oriented database (cơ sở dữ liệu hướng tài liệu) – một thiết kế riêng biệt cho việc lưu trữ document.

Nó là một chương trình máy tính được thiết kế dùng để lưu trữ, truy xuất và quản lý hướng tài liệu hoặc dữ liệu bán cấu trúc hoặc thông tin.

Document-oriented databases là một trong những loại chính của cơ sở dữ liệu NoSQL được biết đến với thuật ngữ “document-oriented document” (hay là “document store”). Trái ngược lại với cơ sở dữ liệu quan hệ truyền thống-quan niệm về “Relations” (hay “Tables”), những hệ thống này được thiết kế xung quanh một khái niệm trừu tượng gọi là “document”.

### Document (tài liệu)

Khái niệm trung tâm của document-oriented database là khái niệm “document”. Trong khi mỗi loại document-oriented database được triển khai không giống nhau ở phần chi tiết so với định nghĩa thì nói chung tất cả đều giả định các tài liệu đóng gói và mã hóa dữ liệu (thông tin) trong một số định dạng tiêu chuẩn hoặc mã hóa. Một số kiểu mã hóa được sử dụng bao gồm XML, YAML, JSON, và BSON, cũng như các hình thức nhị phân như PDF và các tài liệu Microsoft Office (MS Word, Excel …).

Các document (tài liệu) bên trong một document-oriented database thì tương tự nhau,nó gần giống với khái niệm “records” hay “row” trong cơ sở dữ liệu quan hệ truyền thống nhưng nó ít cứng nhắc hơn. Documents không bắt buộc phải tuân theo một lược đồ tiêu chuẩn cũng không cần phải có tất cả các phần, khóa tương tự nhau.

Ví dụ sau đây là một document:

* *FirstName:"Bob", Address:"5 Oak St.", Hobby:"sailing".*

Một document khác có thể là:

* *FirstName:"Jonathan", Address:"15 Wanamassa Point Road", Children:[{Name:"Michael",Age:10}, {Name:"Jennifer", Age:8}, {Name:"Samantha", Age:5}, {Name:"Elena", Age:2}].*

Cả hai document trên có một số thông tin tương tự và một số thông tin khác nhau. Không giống như một cơ sở dữ liệu quan hệ truyền thống, nơi mỗi record(row) có cùng một tập hợp trường dữ liệu (fields hay columns) và các trường dữ liệu này nếu không được sử dụng thì có thể được lưu trữ trống, còn trong một document-oriented database thì không có trường dữ liệu trống trong document. Hệ thống này cho phép thông tin mới được thêm vào và nó không đòi hỏi phải khai báo rõ ràng là khác những mẩu thông tin còn lại.

### Khóa (Keys)

Các document được đề cập trong document-oriented database thông qua một khóa duy nhất đại diện cho documnet đó. Thông thường, khóa này là một chuỗi đơn giản. Trong một số trường hợp, chuỗi này có thể là một URI hoặc đường dẫn (path). Bạn có thể sử dụng khóa này để lấy document (tài liệu)từ cơ sở dữ liệu. Thông thường, cơ sở dữ liệu vẫn lưu lại một chỉ số (index)trong khóa của document để document có thể được tìm kiếm nhanh chóng.

### Lấy thông tin dữ liệu

Một trong các đặc điểm khác biệt của một document-oriented database là việc sử dụng key-document (hoặc key-value) để lấy một tài liệu, cơ sở dữ liệu sẽ cung cấp một API hoặc ngôn ngữ truy vấn cho phép bạn lấy các tài liệu dựa trên nội dung . Ví dụ, bạn muốn truy vấn lấy toàn bộ document mà những document đó có tập hợp trường dữ liệu nhất định với những giá trị nhất định. Tập hợp các API truy vấn hoặc ngôn ngữ truy vấn là tính năng có sẵn, cũng như hiệu năng mong đợi của các truy vấn, thay đổi đáng kể từ một trong những việc triển khai.

### Tổ chức

Các triển khai cung cấp các cách sắp xếp tài liệu, bao gồm khái niệm về:

* *Collections*
* *Tags*
* *Non-visible Metadata*
* *Directory hierarchies*

### Các triển khai của document database

| **Name** | **Publisher** | **License** | **Language** | **Notes** | [**RESTful**](http://en.wikipedia.org/wiki/Representational_State_Transfer)**API** |
| --- | --- | --- | --- | --- | --- |
| [eXist](http://en.wikipedia.org/wiki/EXist) | [eXist](http://en.wikipedia.org/wiki/EXist), [[2]](http://exist-db.org/) | [GPL](http://en.wikipedia.org/wiki/GPL) | [XQuery](http://en.wikipedia.org/wiki/XQuery), [Java](http://en.wikipedia.org/wiki/Java_(programming_language)) | XML over REST/HTTP, WebDAV, Lucene Fulltext search, validation, versioning, clustering, triggers, URL rewriting, collections, ACLS, XQuery Update | Yes |
| [MUMPS](http://en.wikipedia.org/wiki/MUMPS)Database |  | [Proprietary](http://en.wikipedia.org/wiki/Proprietary_software) and GNU Affero GPL | [MUMPS](http://en.wikipedia.org/wiki/MUMPS) | Commonly used in health applications. | (unknown) |
| [Lotus Notes](http://en.wikipedia.org/wiki/Lotus_Notes) | IBM | [Proprietary](http://en.wikipedia.org/wiki/Proprietary_software) | [LotusScript](http://en.wikipedia.org/wiki/LotusScript),[Java](http://en.wikipedia.org/wiki/Java_(programming_language)), Lotus @Formula |  | (unknown) |
| [BaseX](http://en.wikipedia.org/wiki/BaseX) | [BaseX Team](http://basex.org/) | [BSD License](http://en.wikipedia.org/wiki/BSD_License) | [Java](http://en.wikipedia.org/wiki/Java_(programming_language)), [XQuery](http://en.wikipedia.org/wiki/XQuery) | Support for XML, JSON and binary formats; client-/server based architecture; concurrent structural and full-text searches and updates; REST APIs. | Yes |
| [OrientDB](http://en.wikipedia.org/wiki/OrientDB) | [Orient Technologies](http://www.orientechnologies.com/) | [Apache License](http://en.wikipedia.org/wiki/Apache_License) | [Java](http://en.wikipedia.org/wiki/Java_(programming_language)) | JSON over HTTP | Yes |
| [Terrastore](http://en.wikipedia.org/wiki/Terrastore) |  | [Apache License](http://en.wikipedia.org/wiki/Apache_License) | [Java](http://en.wikipedia.org/wiki/Java_(programming_language)) | JSON/HTTP | (unknown) |
| [Jackrabbit](http://en.wikipedia.org/wiki/Apache_Jackrabbit) | [Apache Software Foundation](http://en.wikipedia.org/wiki/Apache_Software_Foundation) | [Apache License](http://en.wikipedia.org/wiki/Apache_License) | [Java](http://en.wikipedia.org/wiki/Java_(programming_language)) |  | (unknown) |
| [CouchDB](http://en.wikipedia.org/wiki/CouchDB) | [Couchbase](http://en.wikipedia.org/wiki/Couchbase),[Apache Software Foundation](http://en.wikipedia.org/wiki/Apache_Software_Foundation) | [Apache License](http://en.wikipedia.org/wiki/Apache_License) | [Erlang](http://en.wikipedia.org/wiki/Erlang_(programming_language)) | JSON over REST/HTTP with [Multi-Version Concurrency Control](http://en.wikipedia.org/wiki/Multi-Version_Concurrency_Control) and limited [ACID](http://en.wikipedia.org/wiki/ACID) properties. Uses[map](http://en.wikipedia.org/wiki/Map_(higher-order_function)) and [reduce](http://en.wikipedia.org/wiki/Fold_(higher-order_function)) for views and queries.[[2]](http://en.wikipedia.org/wiki/Document-oriented_database#cite_note-1) | Yes |
| [FleetDB](http://en.wikipedia.org/wiki/FleetDB) | [FleetDB](http://fleetdb.org/) | [MIT License](http://en.wikipedia.org/wiki/MIT_License) | [Clojure](http://en.wikipedia.org/wiki/Clojure) | A [JSON-based](http://fleetdb.org/docs/protocol.html) schema-free database optimized for agile development. | (unknown) |
| [MarkLogic](http://en.wikipedia.org/wiki/MarkLogic) | MarkLogic Corporation | [Free Express license](http://developer.marklogic.com/licensing) or[Commercial](http://developer.marklogic.com/licensing) | [C++](http://en.wikipedia.org/wiki/C%2B%2B), [XQuery](http://en.wikipedia.org/wiki/XQuery),[XSLT](http://en.wikipedia.org/wiki/XSLT) | Fast, scalable, distributed, enterprise-grade document-oriented database with [Multi-Version Concurrency Control](http://en.wikipedia.org/wiki/Multi-Version_Concurrency_Control), integrated [Full text search](http://en.wikipedia.org/wiki/Full_text_search) and [ACID](http://en.wikipedia.org/wiki/ACID)-compliant transaction semantics | Yes |
| [ThruDB](http://code.google.com/p/thrudb/) |  | [BSD License](http://en.wikipedia.org/wiki/BSD_License) | [C++](http://en.wikipedia.org/wiki/C%2B%2B), [Java](http://en.wikipedia.org/wiki/Java_(programming_language)) | Built on top of [Apache Thrift](http://en.wikipedia.org/wiki/Apache_Thrift) framework that provides indexing and document storage services for building and scaling websites. Alternate implementation is being developed in Java. [Alpha software](http://en.wikipedia.org/wiki/Software_release_life_cycle#Alpha). | (unknown) |
| [Clusterpoint](http://en.wikipedia.org/wiki/Clusterpoint) | [Clusterpoint Ltd.](http://www.clusterpoint.com/) | Free community license /[Commercial](http://en.wikipedia.org/wiki/Commercial_software)[[1]](http://en.wikipedia.org/wiki/Document-oriented_database#cite_note-0) | [C++](http://en.wikipedia.org/wiki/C%2B%2B) | Scalable, high-performance, schema-free, document-oriented [database management system](http://en.wikipedia.org/wiki/Database_management_system)platform with server based data storage, fast [full text search](http://en.wikipedia.org/wiki/Full_text_search) engine functionality, information[ranking](http://en.wikipedia.org/wiki/Ranking) for search relevance and [clustering](http://en.wikipedia.org/wiki/Cluster_computing). | Yes |
| [MongoDB](http://en.wikipedia.org/wiki/MongoDB) | 10gen, Inc | [GNU AGPL v3.0](http://en.wikipedia.org/wiki/Affero_General_Public_License) | [C++](http://en.wikipedia.org/wiki/C%2B%2B) | Fast, document-oriented database optimized for highly transient data. | Optional |
| [RavenDB](http://en.wikipedia.org/w/index.php?title=RavenDB&action=edit&redlink=1) | [Hibernating Rhinos](http://hibernatingrhinos.com/) | [AGPL](http://en.wikipedia.org/wiki/Affero_General_Public_License) Free or[Commercial](http://ravendb.net/licensing/) | [C#](http://en.wikipedia.org/wiki/C_Sharp_(programming_language)), [F#](http://en.wikipedia.org/wiki/F_Sharp_(programming_language)),[VB.NET](http://en.wikipedia.org/wiki/VB.NET) | JSON over REST/HTTP with a rich .Net client API. [ACID](http://en.wikipedia.org/wiki/ACID) compliant. Uses [Lucene.net](http://en.wikipedia.org/wiki/Lucene.net) to provide indexes using a [LINQ](http://en.wikipedia.org/wiki/LINQ) interface, optionally with [MapReduce](http://en.wikipedia.org/wiki/Map_reduce) functionality. | Yes |
| [Redis](http://en.wikipedia.org/wiki/Redis) |  | [BSD License](http://en.wikipedia.org/wiki/BSD_License) | [ANSI C](http://en.wikipedia.org/wiki/ANSI_C) | Key-value store supporting lists and sets with fast, simple and binary-safe protocol. | (unknown) |
| [Rocket U2](http://en.wikipedia.org/wiki/Rocket_U2) | Rocket Software | [Proprietary](http://en.wikipedia.org/wiki/Proprietary_software) |  | UniData, UniVerse | Yes (Beta) |

## Graph Database

Graph database là một dạng cơ sở dữ liệu được thiết kế riêng cho việc lưu trữ thông tin đồ thị như cạnh, nút, các thuộc tính.



Chúng ta có thể nghĩ graph database như một document database với các kiểu document đặc biệt và các mối quan hệ. Một ví dụ điển hình đó chính là mạng xã hội, có thể xem hình bên dưới:



Ví dụ về các nút trong một graph database

Trong ví dụ trên ta có 4 document và 3 mối quan hệ. Mối quan hệ trong graph database thì có ý nghĩa nhiều hơn con trỏ đơn thuần. Một mối quan hệ có thể một chiều hoặc hai chiều nhưng quan trọng hơn là mối quan hệ được phân loại. Một người có thể liên kết với người khác theo nhiều cách, có thể là khách hàng, có thể là người trong gia đình…Mối quan hệ tự bản thân nó có thể mang thông tin. Trong ví dụ ta chỉ đơn giản lưu lại lại của mối quan hệ và mức độ gần gũi.

Với graph database, chúng ta có thể thực hiện các hoạt động đồ thị. Một thao tác cơ bản nhất là traversal (điểm giao nhau). Ví dụ như ta muốn biết những người bạn của ta trong thị trấn để cùng đi ăn uống thì việc này đơn giản. Nhưng còn bạn bè gián tiếp thì sao, làm sao ta biết được họ. Sử dụng graph database chúng ta có thể định nghĩa truy vấn sau:

|  |
| --- |
| *new GraphDatabaseQuery*  *{*  *SourceNode = ayende,*  *MaxDepth = 3,*  *RelationsToFollow = new[]{"As Known As", "Family", "Friend", "Romantic", "Ex"},*  *Where = node => node.Location == ayende.Location,*  *SearchOrder = SearchOrder.BreadthFirst*  *}.Execute();* |

Chúng ta có thể thực hiện những truy vấn phức tạp hơn như lọc trên các thuộc tính quan hệ, xem xét trọng lượng của người đó... Graph database thường được sử dụng để giải quyết các vấn đề về mạng. Trong thực tế, hầu hết các trang web mạng xã hội đều sử dụng một số hình thức của graph database để làm những việc mà chúng ta đã biết như: kết bạn, bạn của bạn…

Một vấn đề đối với việc mở rộng graph database là rất khó để tìm thấy một đồ thị con độc lập, có nghĩa là rất khó để ta phân tán graph database thành nhiều mảnh. Có rất nhiều nỗ lực nghiên cứu cho việc này nhưng chưa có bất kỳ giải pháp nào đáng tin cậy được đưa ra.

Một số sản phẩm tiêu biểu của graph database là: Neo4J, Sones, AllegroGraph, Core Data, DEX, FlockDB, InfoGrid, OpenLink Virtuoso,...

| **Name** | **Language** | **Notes** |
| --- | --- | --- |
| [AllegroGraph](http://en.wikipedia.org/wiki/AllegroGraph) | [SPARQL](http://en.wikipedia.org/wiki/SPARQL) | [RDF](http://en.wikipedia.org/wiki/Resource_Description_Framework) GraphStore |
| [DEX](http://en.wikipedia.org/wiki/DEX_(Graph_database)) | [Java](http://en.wikipedia.org/wiki/Java_(programming_language)), [C++](http://en.wikipedia.org/wiki/C%2B%2B) | High-performance [Graph Database](http://en.wikipedia.org/wiki/Graph_Database) |
| [FlockDB](http://en.wikipedia.org/wiki/FlockDB) | [Scala](http://en.wikipedia.org/wiki/Scala_(programming_language)) |  |
| [InfiniteGraph](http://en.wikipedia.org/wiki/InfiniteGraph) | [Java](http://en.wikipedia.org/wiki/Java_(programming_language)) | High-performance, scalable, distributed [Graph Database](http://en.wikipedia.org/wiki/Graph_Database) |
| [Neo4j](http://en.wikipedia.org/wiki/Neo4j) | [Java](http://en.wikipedia.org/wiki/Java_(programming_language)) |  |
| [OpenLink Virtuoso](http://en.wikipedia.org/wiki/Virtuoso_Universal_Server) | [C++](http://en.wikipedia.org/wiki/C%2B%2B), [C#](http://en.wikipedia.org/wiki/C_Sharp_(programming_language)), [Java](http://en.wikipedia.org/wiki/Java_(programming_language)), [SPARQL](http://en.wikipedia.org/wiki/SPARQL) | [middleware](http://en.wikipedia.org/wiki/Middleware) and [database engine](http://en.wikipedia.org/wiki/Database_engine) hybrid |
| [OrientDB](http://en.wikipedia.org/wiki/OrientDB) | [Java](http://en.wikipedia.org/wiki/Java_(programming_language)) |  |
| [Pregel](http://en.wikipedia.org/wiki/Pregel) |  |  |
| [Sones GraphDB](http://en.wikipedia.org/wiki/Sones_GraphDB) | [C#](http://en.wikipedia.org/wiki/C_Sharp_(programming_language)) |  |
| [OWLIM](http://en.wikipedia.org/wiki/Ontotext) | [Java](http://en.wikipedia.org/wiki/Java_(programming_language)), [SPARQL 1.1](http://en.wikipedia.org/wiki/SPARQL) | [RDF](http://en.wikipedia.org/wiki/Resource_Description_Framework) graph store with reasoning |

## Làm sao để lựa chọn một giải pháp cơ sở dữ liệu

Như các phần trên đã đề cập đến các giải pháp cơ sở dữ liệu NoSQL thì mỗi loại trong số đó có những điểm mạnh và điểm yếu riêng của nó. Một câu hỏi chúng ta thường hay gặp là: Tôi muốn sử dụng công nghệ NoSQL X cho việc XYZ…

Với câu hỏi này, chúng ta thường gặp phải vấn đề là:

* Cố gắng áp dụng các khái niệm, kỹ thuật, kinh nghiệm của mô hình cơ sở dữ liệu quan hệ vào trong NOSQL.
* Cố gắng sử dụng một loại cơ sở dữ liệu NoSQL cho tất cả mọi thứ bao gồm cả thứ mà nó thực sự không phù hợp.

Lựa chọn một chiến lược lưu trữ cơ sở dữ liệu không phải quyết định một lần là được. Trong một ứng dụng, chúng ta có thể sử dụng key-value store để lưu trữ thông tin phiên làm việc (session), sử dụng graph database để phục vụ những truy vấn xã hội và document database để lưu trữ các thực thể. Nếu chúng ta lưu trữ dữ liệu theo một loại cơ sở dữ liệu NoSQl duy nhất thì việc này giống như chúng ta muốn lưu trữ tất cả code trên một file duy nhất. Chúng ta có thể làm được việc này nhưng có vẻ vụng về, không được tối ưu lắm. Điều nên làm là cố gắng phân chia ứng dụng thành từng phần mà mỗi phần thích hợp với một mô hình truy cập dữ liệu để đem lại hiểu quả cao nhất. Ví dụ như trong danh mục sản phẩm luôn làm việc với những truy vấn bởi Product SKU và tốc độ là cốt yếu thì ta nên sử dụng key-value store. Nhưng điều đó không có nghĩa là đơn đặt hàng cũng được lưu trữ ở đó mà chúng ta cần tính linh hoạt hơn nên sẽ sử dụng document database…

Kết luận lại là trong một ứng dụng chúng ta có thể sử dụng nhiều công nghệ lưu trữ dữ liệu khác nhau để làm cho ứng dụng của chúng ta hoạt động tốt nhất và mỗi phần khác nhau của ứng có thể sử dụng công nghệ khác nhau sao cho phù hợp với mục đích của chúng ta. Điều đó cũng nói lên rằng là một công nghệ lưu trữ dữ liệu mới chỉ thực sự có nghĩa khi mà lợi ích nó mang lại trong hệ thống có nhiều công nghệ lưu trữ dữ liệu lớn hơn chi phí phải trả. Nếu chúng ta cần hỗ trợ lưu trữ các trường dữ liệu người dùng tự định nghĩa thì chúng ta nhanh chóng sử dụng document database hơn là cố gắng thực hiện điều đó với RDBMs.

Lưu ý: Không nên quên RDMBs. Mặc dù với cái tên NoSQL thì NoSQL thực sự là viết tắt của Not only SQL(không chỉ SQL). Vấn đề chính không phải là công nghệ RDBMs mà vấn đề là đối với nhiều người, lưu trữ dữ liệu là RDBMs. Khi chọn công nghệ lưu trữ dữ liệu chúng ta cần quan tâm tới việc kết hợp với RDBMs. RDBMs là một công cụ rất mạnh mẽ và không nên được bỏ đi chỉ vì đối thủ còn non trẻ và hấp dẫn hơn.

Trong khóa luận tốt nghiệp này, chúng tôi chọn Document Database làm cơ sở dữ liệu NoSQL để xây dụng ứng dụng chính. Những lý do mà chúng tôi chọn Document Store là:

* Về cơ bản thì cốt lõi của Document Database là key-value store được lưu trữ theo một định dạng được biết đến. Do đó, document database cũng đáp ứng được yêu cầu của một key-value store khi cần truy cập dữ liệu theo khóa.
* Document database yêu cầu dữ liệu được lưu trữ dưới định dạng mà cơ sở dữ liệu hiểu được. Các định dạng có thể là XML, JSON, Binary JSON(BSON) miễn sao cơ sở dữ liệu hiểu được cấu trúc nội bộ của document. Thực tế thì hầu hết các ứng đều sử dụng JSON (hoặc BSON) hoặc XML. Đây đều là những định dạng được sử dụng rất phổ biến và con người có thể đọc được.
* Cơ sở dữ liệu hiểu được định dạng của dữ liệu thì nó có thể thực hiện thao tác trên dữ liệu này phía máy chủ. Các định dạng này được biết đến điều đó có nghĩa là nó dễ dàng hơn để viết các công cụ quản lý dữ liệu vì có thể hiển thị và chỉnh sữa dữ liệu.
* Document database có lược đồ tùy ý. Chúng ta không cần phải định nghĩa trước lược đồ và tuân thủ theo lược đồ này. Điều này cho phép chúng ta lưu trữ dữ liệu phức tạp tùy ý. Có thể lưu trữ dữ liệu dạng cây, tập hợp hay dạng từ điển một cách dễ dàng.
* Thực tế, lợi ích chính của việc sử dụng document database là ngoài việc nó có tất cả lợi ích của key-value store thì chúng ta không bị giới hạn bởi việc truy vấn theo khóa. Bằng cách lưu trữ dữ liệu theo định dạng được biết đến mà cơ sở dữ liệu có thể hiểu được, chúng ta có thể yêu cầu máy chủ làm việc chẳng hạng như truy vấn. Ví dụ, các yêu cầu HTTP sau sẽ tìm thấy tất cả tài liệu có tên là Ayende:

|  |
| --- |
| *GET /indexes/dynamic?query=name:ayende* |

* Bên cạnh việc có thể truy vấn dữ liệu, document database còn có thể:
  + Thực hiện phép chiếu dữ liệu của một document sang một định dạng khác.
  + Chạy phép tính tập hợp trên một tập hợp các document.
  + Cập nhật một phần dữ liệu (có nghĩa chúng ta không cần load lên toàn bộ một thực thể, thay đổi và lưu xuống lại)
* Lợi ích rất quan trọng của việc sử dụng document database là làm việc với các documents. Có rất ít hoặc không có trở kháng không phù hợp giữa đối tượng và document. Điều này có nghĩa là việc lưu trữ dữ liệu trong document database sẽ dễ dàng hơn rất nhiều so với việc sử dụng RDBMS trong hầu hết các kịch bản mà dữ liệu cần lưu trữ không đơn giản. Chúng ta thường khá vất vả để thiết kế mô hình dữ liệu vật lý trong RDBMS bởi vì cách chúng ta đặt dữ liệu trong cơ sở dữ liệu và cách chúng ta nghĩ về nó trong ứng dụng hoàn toàn khác nhau. Hơn nữa trong RDBMS còn có khái niệm lược đồ và sửa đổi lược đồ là một điều thực sự khó khăn nếu chúng ta triển khai trên nhiều node của hệ thống.
* Document không hỗ trợ mối quan hệ. Điều đó có nghĩa là mỗi document là độc lập và chúng ta sẽ dễ dàng phân tán dữ liệu hơn so với RDBMS bởi vì chúng ta không cần lưu trữ tất cả các quan hệ trên cùng một mảnh của hệ thống và không cần hỗ trợ phép join trên hệ thống phân tán.

Trong chương tiếp theo, chúng ta sẽ tìm hiểu về một số loại document database phổ biến như: MongoDB, CouchDB và RavenDB. Sau đó sẽ đi sâu vào tìm hiểu RavenDB, một triển khai của document database rất phổ biến dành cho nền tảng .Net.

# CHƯƠNG 4: TÌM HIỂU VỀ RAVENDB

## Một số triển khai phổ biến của document database

### MongoDB

Mongo là một cơ sở dữ liệu NoSQL nguồn mở, hiệu năng cao, có tính mở rộng cao và được viết bằng C++. Mongo dùng cách lưu trữ BSON (Json được biên dịch) với giấy phép AGPL.Thay vì lưu trữ dữ liệu theo các bảng như cơ sở dữ liệu truyền thống, MongoDB lưu trữ cấu trúc dữ liệu thành các văn bản dựa JSON với mô hình động (gọi đó là BSON) khiến cho việc tích hợp dữ liệu cho các ứng dụng trở nên dễ dàng và nhanh hơn.Với mục tiêu là kết hợp các điểm mạnh của mô hình key-value (nhanh mà tính mở rộng cao) với mô hình dữ liệu quan hệ ( giàu chức năng).

Mục tiêu chính của Mongo là giữ lại các thuộc tính thân thiện của SQL.Do đó các câu truy vấn khá giống với SQL nên MongoDB khá thích hợp cho các lập trình viên đã quen với ngôn ngữ truy vấn SQL. MongoDB có nhiều tính năng và hiệu năng cao. Với các loại dữ liệu phong phú, nhiều truy vấn và việc giảm thời gian phát triển trong việc mô hình hóa các đối tượng.

MongoDB được sử dụng tốt nhất với nhu cầu cần truy vấn động, nếu chúng ta muốn định nghĩa chỉ mục mà không cần các hàm map/reduce. Đặc biệt nếu chúng ta cần tốc độ nhanh cho một cơ sở dữ liệu lớn vì MongoDB ngoài tốc độ đọc nhanh ra thì tốc độ ghi của nó rất nhanh.

Các đặc điểm chính của mongoDB là:

* Các truy vấn Ad hoc: Mongo hỗ trợ việc tìm theo trường dữ liệu, khoảng kết quả tìm và tìm theo cú pháp. Các truy vấn có thể trả về các trường được qui định trong document và cũng có thể bao gồm các hàm Javascript mà người dùng chưa định nghĩa.
* Đánh chỉ mục: Bất cứ một trường dữ liệu nào trong MongoDB đều được đánh chỉ mục (giống như chỉ mục bên RMDBs).
* Nhân bản: Mongo hỗ trợ nhân bản kiểu Master-slave. Một master có thể điều khiển việc đọc và ghi. Một slave tạo bản sao dữ liệu từ master và chỉ được sử dụng cho việc đọc và backup (không có quyền ghi). Slave có khả năng chọn ra một master mới nếu master cũ bị hỏng.
* Cân bằng tải: Mongo mở rộng theo chiều ngang bằng cách sử dụng sharding. Các lập trình viên chọn các khóa chia sẻ nhằm xác định dữ liệu sẽ được phân tán như thế nào.Dữ liệu sẽ được tách thành các khoảng dựa vào khóa và phân tán dọc theo các Shard.
* Lưu trữ file: Mongo lưu trữ bằng file hệ thống, rất tốt cho việc cân bằng tải và nhân bản dữ liệu. Trong các hệ thống nhiều máy, các file được phân phối và được sao chép ra rất nhiều lần giữa các máy một cách trong suốt. Do đó rất hiệu quả trong việc tạo ra một hệ thống cân bằng tải và dung lỗi tốt.

### CouchDB

Được viết bằng Erlang với mục tiêu là tạo ra một cơ sở dữ liệu bền vững, chịu lỗi cao, dễ dàng trong việc sử dụng. Dùng cách lưu trữ thông thường là JSON với giấy phép Apache 2.0.

Với CouchDB thì mỗi một cơ sở dữ liệu là một tập các document riêng biệt. Mỗi document tự bảo quản chính nó và tự nó bao gồm mô hình của nó ( các trường dữ liệu, loại của mỗi trường). Mỗi một ứng dụng có thể thực thi rất nhiều cơ sở dữ liệu, ví dụ như chúng ta dùng một cơ sở dữ liệu để lưu thông tin người dùng điện thoại và cái còn lại là lưu trên máy chủ. Trên mỗi document còn bao gồm các thông tin về phiên bản, dễ dàng đồng bộ các dữ liệu với nhau khi cơ sở dữ liệu bị mất kết nối một thời gian giữa các thiết bị.

CouchDB dử dụng MVCC (multi-Version Concurency Control ) để tránh việc deadlock database trong suốt quá trình ghi. Tức là nếu trong quá trình ghi dữ liệu, chúng ta vẫn có thể đọc dữ liệu vì CouchDB sinh ra một bản sao và chúng ta đọc trên bản sao đó. Sau khi ghi xong nó sẽ tiến hành nhập dữ liệu giữa các thiết bị và xóa bản ghi cũ đi.

Dùng giao thức HTTP theo Restful với cách thiết kế có khả năng chịu lỗi cao với dùng views đi kèm với map/reduce mang lại một tốc độ cao.Thích hợp cho rất nhiều các thiết bị khác nhau như máy chủ, máy bàn hay điện thoại thông minh. CouchDB được sử dụng tốt nhất cho các hệ thống thỉnh thoảng thay đổi dữ liệu như các hệ thống CMS, các hệ thống cho phép triển khai nhiều trang web.

Các đặc điểm chính của CouchDB:

* Lưu trữ theo hướng tài liệu (document storage)
* Sử dụng ngữ nghĩa ACID: Cho phép điều khiển việc đồng bộ việc ghi và đọc cường độ rất cao mà không lo bị xung đột.
* Sử dụng Map/Reduce và các chỉ mục: Mỗi view được tạo ra bởi một hàm javascript mà thực thi cả 2 hành động map và reduce. Hàm đó khiến cho các document kết hợp với nhau thành một giá trị đơn nhất và trả về kết quả đó.
* Kiến trúc phân tán có nhân bản: CouchDB được thiết kế với khả năng nhân bản 2 chiều với các dữ liệu offline. Tức là ta có thể chỉnh sửa dữ liệu offline và sau đó đồng bộ chúng sau khi có kết nối trở lại.
* REST API: Tất cả dữ liệu đều có một địa chỉ duy nhất được lấy qua HTTP. Giao thức REST sử dụng các phương thức của HTTP như GET, POST, PUT và DELETE với 4 chức năng cơ bản (Tạo, đọc, ghi, xóa, sửa).
* Built for Offline: Có khả năng nhân bản dữ liệu cho từng thiết bị và tự động đồng bộ dữ liệu khi thiết bị hoạt động trở lại.

### RavenDB



RavenDB được viết trên C# bởi Hibernating Rhinos với giấy phép GNU AGPL v3.0. RavenDB là một giải pháp NoSQL trên nền tảng .NET.

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.

Các đặc điểm chính của RavenDB:

* Mặc định an toàn dữ liệu: Hỗ trợ ACID (Atomicity, Consistency, Isolation, Durability), No locking, Automatic batching, client/server chatter projection.
* .Net client API: hỗ trợ tốt cho việc lập trình trên nền tảng .NET
* REST API: Tất cả dữ liệu đều có một địa chỉ duy nhất được lấy qua HTTP. Giao thức REST sử dụng các phương thức của HTTP như GET, POST, PUT và DELETE.
* Dễ dàng triển khai ứng dụng một cách nhanh chóng (chưa đến 5 phút)
* Kiến trúc phân tán: mở rộng ứng dụng một cách dễ dàng bằng cách sử dụng tính năng mạnh mẽ của RavenDB cho việc mở rộng là Sharding và Replication. Có thể kết hợp cả hai tính năng này trong cùng ứng dụng. Có hỗ trợ multi-database.
* Hỗ trợ nhiều gói tiện ích hữu dụng như: Versioning, Expiration, IndexReplication, Authorization, Authentication. Chúng ta có thể tự viết các gói mở rộng cho RavenDB bằng cách sử dụng Triggers và Responders.

## Tại sao chọn RavenDB

RavenDB là một document database nên nó thừa hưởng những lợi ích to lớn của cơ sở dữ liệu NoSQL nói chung và cơ sở dữ liệu hướng tài liệu nói riêng. Những lợi ích to lớn này chúng ta đã đề cập ở những phần trên. Ngoài ra, RavenDB còn có những đặc điểm, tính năng nổi bật khác như sau:

* RavenDB là một cơ sở dữ liệu hướng tài liệu mã nguồn mở có hỗ trợ transactional được viết cho nền tảng .NET. RavenDB đưa ra mô hình dữ liệu linh hoạt nhằm đáp ứng yêu cầu của các hệ thống thế giới thực. RavenDB cho phép xây dựng những ứng dụng có hiệu suất cao, độ trễ thấp 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 một cách 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 tán dữ liệu thành các phần nhỏ lưu trên nhiều server khác nhau).
* 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 RavenDB vào trong ứng dụng.
* Hỗ trợ System.Transaction và có thể thực hiện các transactions trong hệ thống phân tán.
* 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.
* 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.
* Thích hợp cho cả sản phẩm mã nguồn mở và các sản phẩm thương mại.

## Tính năng đầy đủ của RavenDB

* **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**

Hỗ trợ đầy đủ ACID transactions (Atomicity, Consistency, Isolation, Durability) ngay cả những node khác nhau trong hệ thống.

* **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**

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**

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 Work**
* **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**
  + 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**

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**

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**

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 managemet studio**

Dễ dàng quản lý dữ liệu với giao diện đồ họa trực quan.

* **Carefully designed**

RavenDB được thiết kế rất cẩn thận, tỉ mĩ đảm bảo mọi thứ hoạt động tốt.

* **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**

Hỗ trợ nhiều tính năng và khả năng mở rộng.

* **Embededable**

RavenDB có thể nhúng vào bất kỳ ứng dụng .NET, và nó cũng hoàn toàn phù hợp với các ứng dụng desktop.

* **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**

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**

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.

* **Advances search techniques**
* **Geo-spatial search support**

Dễ dàng sử dụng API này.



* **Easy backup**

Việc 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**

Lưu trữ nhiều database trên một RavenDB Server.



* **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 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 (C# 5 ready)**

RavenDB hỗ trợ API bất đồng bộ mới được giới thiệu bởi C#5

* **Community**
* **Cloud hosting available**

Chạy RavenDB trên đám mây với RavenHQ, CloudBird, AppHorbor hoặc Windows Azure.



## Cơ sở lý thuyết RavenDB

### Lý thuyết cơ bản 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, chúng ta thường 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 document 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. Collection là một cấu trúc hoàn toàn ảo, không có ý nghĩa vật lý đối với cơ sở dữ liệu.

Với RavenDB mỗi document 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:



#### Tạo khóa cho các document

RavenDB hỗ trợ các cách tạo khóa cho các document sau:

##### 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 tại 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) (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 cache cho server.

#### Kết nối tới RavenDB 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 RavenDB 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"/>*  *<add name="Secure" connectionString="Url = 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:

|  |
| --- |
| *// lưu tất cả thay đổi sử dụng session API*  *using (IDocumentSession session = store.OpenSession())*  *{*  *// Sử dụng session để thao tác với cơ sở dữ liệu*    *// 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 đầu 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:

|  |
| --- |
| *// tạo thực thể mới của lớp BlogPost*  *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()

|  |
| --- |
| *// lưu dữ liệu xuống 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 Id này sẽ được lấy giá trị bằng cách gọi hàm 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 là một thực thể của collection BlogPost với Id là 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 dữ liệu

Để đáp ứng yêu cầu trong khoảng thời gian rất nhanh, máy chủ RavenDB index dữ liệu của chúng ta bên dưới nền bất kì khi nào nó được thêm hay thay đổi. Tất cả indexes trong RavenDB đều dựa trên Lucene và chúng ta tận dụng những ưu điểm này để cung cấp hệ thống truy vấn một cách nhanh chóng, đầy đủ tính năng và linh hoạt. RavenDB cho phép sử dụng cú pháp Lunce để truy vấn cho dù chúng được gửi từ Client API thông qua Linq provider hoặc thông qua HTTP RESTful API.Chúng được chuyển sang truy vấn Lucene và thực thi dựa vào những index thích hợp.

Hầu hết các thao tác này đều được thực hiện trong suốt (chúng ta không nhìn thấy được). Tất cả chúng ta cần là làm quen với Linq và một số thao tác cụ thể với RavenDB. Tuy nhiên còn có rất nhiều tính năng khác hữu dụng khi chúng ta hiểu cách làm việc của RavenDB thao tác trên những indexes.

Một điều quan trọng cần lưu ý là tất cả truy vấn tới RavenDB server đều sử dụng index để trả về kết quả. Chúng ta có thể định nghĩa index riêng cho chúng ta (sẽ nói ở phần tiếp theo), còn không thì RavenDB sẽ làm chuyện đó một cách tự động.

Có 2 loại index trong RavenDB:

* Static index là index mà do chính người dùng định nghĩa
* Dynamic index là index được RavenDB tự động tạo ra dựa trên truy vấn cả người dùng nếu không có truy vấn nào khớp với yêu cầu. RavenDB sẽ tìm index thích hợp để truy vấn và sẽ tạo index kèm với truy vấn nếu nó không tồn tại. RavenDB sẽ tối ưu hóa trên các yêu cầu thực tế và có thể quyết định chuyển một index tạm thời thành một index được lưu trữ trên server.

Một khái niệm cũng cần nói đến là khái niệm “stale index”. Bởi vì phương pháp tiếp cận của RavenDB là “better stale than offline” – truy vấn index có thể trả về kết quả cũ. Ví dụ, khi người dùng truy vấn dữ liệu trong khi có quá trình cập nhật một lượng lớn dữ liệu. RavenDB sẽ thông báo cho người dùng biết được nếu kết quả đó là cũ và cũng có thể cho biết phải chờ đợi cho đến khi kết quả không cũ(non-stale result).

##### 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.

##### 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.

#### 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 1: Denormalization (Phi chuẩn hóa)

Cách dễ nhất để giải quyết vấn đề này là phi chuẩn hóa dữ liệu (denormalization data) vào trong thực thể chứa. 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.

Sử dụng 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.

#### 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.

RESTful là mục đích của HTTP API nhưng chỉ là mục đích thứ yếu so với mục đích trình bày dễ dàng để sử dụng những tính năng mạnh mẽ như batching và multi-document transactions.

### Phía máy chủ RavenDB

#### Mở rộng hệ thống theo chiều ngang( scaling out hay là scaling horizontally)

RavenDB hỗ trợ sẵn 2 gói mở rộng là nhân bản(Replication) và phân tán dữ liệu trên nhiều máy chủ (sharding). Cả hai tính năng này trực giao, có nghĩa là chúng có thể được sử dụng kết hợp với nhau.

Sharding là một tính năng phía máy khách, có nghĩa là toàn bộ quyết định được thực hiện phía máy khách. Nhân bản được thực hiện giữa 2 điểm đầu, các máy chủ tự thực hiện việc nhân bản và cũng cần nhận biết các trường hợp khi một trong các nút (node) bị hư hỏng và khi một nút báo cáo lỗi xung đột khi nhân bản.

##### Replication

RavenDB hỗ trợ nhân bản. Để sử dụng chúng thì ta cần phải đính kèm gói Replication khi chúng ta tạo ra một database mới.

Sử dụng Replication sẽ có những tác dụng trên hệ thống của chúng ta:

* Theo dõi trên máy chủ những document được viết lên lần đầu(bản gốc). Gói nhân bản sử dụng thông tin này để xác định một document được nhân bản có bị xung đột với ducoment đã tồn tại hay không.
* Những document gặp phải lỗi xung đột sẽ được đánh dấu thích hợp và sẽ được thực hiện tự động hoặc có sự tham gia của người dùng để giải quyết.
* Những document sẽ được xóa khi có đánh dấu xóa và gói nhân bản cũng dùng dấu hiệu này để xóa những document có liên quan. Đây là một chi tiết được thực hiện mà không cần lưu ý cho máy khách.
* Một vài thiết bị đầu cuối bắt đầu đáp ứng nhưng không giới hạn /replication/replicate và /replication/lastEtag.
* Gói nhân bản sẽ không sao chép một số tài liệu hệ thống(có key bắt đầu bằng Raven/)

Gói nhân bản sẽ tạo ra một số tài liệu hệ thống, không giới hạn Raven/Replication/Destinations – danh sách máy chủ chúng ta cần nhân bản và Raven/Replication/Sources/[server] – thông tin về dữ liệu được sao chép từ một máy chủ cụ thể.

Các máy chủ cần nhân bản tài liệu:

* Các tài liệu đích được lưu với ID của Raven/Replication/Destinations và nó thông báo cho RavenDB biết những máy cần được nhân bản. Định dạng như sau:

|  |
| --- |
| *{*  *"Destinations": [*  *{*  *"Url": "http://raven\_two:8080/"*  *},*  *{*  *"Url": "http://raven\_three:8080/"*  *},*  *]*  *}* |

* Với một đối tượng có chứa một địa chỉ cho mỗi máy chủ được nhân bản. Bất cứ khi nào một document được cập nhật, nhân bản được bắt đầu và bắt đầu sao chép danh sách điểm đến được cập nhật.

**Làm thế nào để nhân bản hoạt động?**

Đối với mỗi thực hiện giao dịch, Raven sẽ tìm kiếm danh sách các điểm đến để nhân bản. Với mỗi điểm đến, gói nhân bản sẽ:

* Truy vấn thể hiện từ xa cho các document cuối cùng mà chúng ta nhân bản cho các thể hiện đó.
* Bắt đầu gửi một tập các cập nhật đã xảy ra kể từ lần cuối cùng được nhân bản. Việc nhân bản thực hiện dưới nền và thực hiện song song.

**Những lỗi khi nhân bản?**

Bởi vì cách nó được thiết kế, một nút có thể bị hư hỏng trong một thời gian dài và bắt đầu chấp nhận mọi thứ bị mất trong khi đó. Gói nhân bản theo dõi một mục dữ liệu duy nhất cho mỗi máy chủ nhân bản, giá trị etag cuối cùng được nhìn thấy từ máy chủ đó. Điều này có nghĩa là chúng ta không phải lo lắng về việc thiếu các cửa sổ sao chép.

##### Sharding

RavenDB hỗ trợ sẵn phân tán dữ liệu. Sharding là cách chia dữ liệu của chúng ta trên nhiều máy chủ khác nhau. Vì thế mỗi máy chủ sẽ lưu trữ một phần dữ liệu. Điều này là cần thiết trong trường hợp chúng ta cần xử lý một lượng lớn dữ liệu.

Giả sử trong một ứng dụng cần xử lý dữ liệu từ rất nhiều công ty trên khắp thế giới thì một sự lựa chọn sẽ là lưu trữ dữ liệu của một công ty trên một mảnh của hệ thống (gọi là shard) và việc lưu trữ này phụ thuộc vào vị trí khu vực (Region) của công ty. Ví dụ, những công ty nằm ở châu Á(Asia) sẽ được lưu trữ trên một mảnh, những công ty nằm ở Trung Đông(Middle East) sẽ được lưu trữ trên một mảnh khác và những công ty từ Mỹ sẽ được lưu trữ vào một mảnh thứ ba.

Ý tưởng trên là xác định vị trí địa lý của mảnh gần nơi mà dữ liệu được sử dụng. Do đó các công ty ở châu Á sẽ được phục vụ từ một máy chủ gần đó và phản hồi nhanh hơn cho người dùng nằm ở châu Á. Điều này cũng làm giảm tải trên mỗi máy chủ bởi vì nó chỉ xử lý một số phần của dữ liệu.

RavenDB có hỗ trợ sẵn sharding. Nó sẽ quản lý tất cả việc sharding cho chúng ta và nhiệm vụ duy nhất của chúng ta là xác định những chức năng của mảnh (làm thế nào để thực sự phân chia các tài liệu giữa các mảnh khác nhau).

Dưới đây là một số thực thể có thể được chia vào các mảnh khác nhau dựa trên khu vực của nó: Company và Invoice

|  |
| --- |
| *public class Company*  *{*  *public string Id { get; set; }*  *public string Name { get; set; }*  *public string Region { get; set; }*  *}*    *public class Invoice*  *{*  *public string Id { get; set; }*  *public string CompanyId { get; set; }*  *public decimal Amount { get; set; }*  *public DateTime IssuedAt { get; set; }*  *}* |

Để làm được điều này, chúng ta cần sử dụng ShardedDocumentStore thay cho DocumentStore thông thường. Ngoại trừ giai đoạn khởi tạo, ShardedDocumentStore cũng dùng tương tự như DocumentStore thông thường và chúng ta có quyền truy cập tất cả các API thông thường và các tính năng của RavenDB.

Để tạo một ShardedDocumentStore chúng ta nên cung cấp một thể hiện của ShardStrategy – chứa một từ điển những mảnh hoạt động trên đó. Các khóa và giá trị trong từ điển là ID của mảnh (shardID) và một thể hiện DocumentStore trên mảnh đó.

ShardStrategy thông báo cho ShardedDocumentStore biết làm thế nào để tương tác với những mảnh của hệ thống. Điều này cho phép chúng ta tùy chỉnh một số khía cạnh khác nhau về các hành vi phân tán, cho chúng ta tùy chọn kiểm soát mảnh tốt hơn như thế nào mà khi RavenDB xử lý dữ liệu của chúng ta:

* ShardAccessStrategy: một thể hiện mà hiện thực interface IShardAccessStrategy cho biết làm thế nào để liên hệ với mảnh đó. SequentialShardAccessStrategy và ParallelShardAccessStrategy là 2 hiện thực của interface này cho phép chúng ta truy cập mảnh theo tuần tự hay là song song. Giá trị mặc định là SequentialShardAccessStrategy.
* ShardResolutionStrategy: một thể hiện mà hiện thực interface IShardResolutionStrategy. Nó quyết định những mảnh nào được liên hệ để hoàn thành thao tác cơ sở dữ liệu. Hiện thực mặc định là lớp DefaultShardResolutionStrategy cho phép chúng ta bắt đầu phân tán dữ liệu nhanh chóng mà không cần hiện thực một IShardResolutionStrategy mới.
* MergeQueryResults: một delegate cho phép chúng ta quyết định làm thế nào để hợp nhất các truy vấn từ nhiều mảnh khác nhau. Có một hiện thực mặc định cho việc này sẽ kết hợp các kết quả lại và áp dụng phân loại hành vi tối thiểu.
* ModifyDocumentId: cho phép chúng ta lưu trữ shard id cho một document trong chính document đó. Mặc định là:

|  |
| --- |
| *(convention, shardId, documentId) => shardId + convention.IdentityPartsSeparator + documentId* |

Để sử dụng sharding, chúng ta có thể chỉ sử dụng ShardStrategy với hành vi mặc định của nó:

|  |
| --- |
| *var shards = new Dictionary<string, IDocumentStore>*  *{*  *{"Asia", new DocumentStore {Url = "*[*http://localhost:8080*](http://localhost:8080/)*"}},*  *{"Middle East", new DocumentStore {Url = "*[*http://localhost:8081*](http://localhost:8081/)*"}},*  *{"America", new DocumentStore {Url = "*[*http://localhost:8082*](http://localhost:8082/)*"}},*  *};*    *var shardStrategy = new ShardStrategy(shards)*  *.ShardingOn<Company>(company => company.Region)*  *.ShardingOn<Invoice>(x => x.CompanyId);*    *var documentStore = new ShardedDocumentStore(shardStrategy).Initialize();* |

Đối với môi trường sharding phức tạp chúng ta có thể hiện thực IShardResolutionStrategy cho riêng mình và thiết lập thuộc tính ShardResolutionStrategy trong ShardStrategy để giữ những tùy chỉnh thay vì dùng DefaultShardResolutionStrategy. IShardResolutionStrategy có một số phương thực cần được hiện thực:

* GenerateShardIdFor: chúng ta có thể quyết định mảnh nào nên được sử dụng để lưu trữ một thực thể cụ thể.
* MetadataShardIdFor: chúng ta có thể quyết định mảnh nào nên được sử dụng để lưu trữ các tài liệu siêu dữ liệu (như các tài liệu Hilo) cho một thực thể cụ thể.
* PotentialShardsFor: chúng ta có thể quyết định mảnh nào nên được liên lạc để hoàn thành một hoạt động truy vấn. Chúng ta có thể quyết định việc này dựa trên các thông số có sẵn có thể là DocumentKey, EntityType và the Query.
* Mặc định nếu chúng ta không thiết lập thuộc tính ShardResolutionStrategy trên ShardStrategy thì chúng ta sẽ sử dụng DefaultShardResolutionStrategy.

Default Shard Resolution Strategy:

* Nếu sử dụng DefaultShardResolutionStrategy chúng ta có thể sử dụng phương thức ShardingOn trên đối tượng ShardStrategy để chỉ cho DefaultShardResolutionStrategy biết thuộc tính nào lưu giữ thông tin shard id của một thực thể cụ thể.
* Đoạn code bên dưới cho thấy Company giữ shard id trong thuộc tính Region và Invoice giữ shard id trong thuộc tính CompanyId. Thuộc tính CompanyId giữ shard id theo quy tắc ModifyDocumentId của ShardStrategy.

|  |
| --- |
| *var shardStrategy = new ShardStrategy(shards)*  *.ShardingOn<Company>(company => company.Region)*  *.ShardingOn<Invoice>(x => x.CompanyId);* |

Giờ chúng ta có thể lưu trữ dữ liệu trên nhiều mảnh khác nhau:

|  |
| --- |
| *using (var session = documentStore.OpenSession())*  *{*  *var asian = new Company { Name = "Company 1", Region = "Asia" };*  *session.Store(asian);*  *var middleEastern = new Company { Name = "Company 2", Region = "Middle-East" };*  *session.Store(middleEastern);*  *var american = new Company { Name = "Company 3", Region = "America" };*  *session.Store(american);*    *session.Store(new Invoice { CompanyId = american.Id, Amount = 3, IssuedAt = DateTime.Today.AddDays(-1) });*  *session.Store(new Invoice { CompanyId = asian.Id, Amount = 5, IssuedAt = DateTime.Today.AddDays(-1) });*  *session.Store(new Invoice { CompanyId = middleEastern.Id, Amount = 12, IssuedAt = DateTime.Today });*  *session.SaveChanges();*  *}* |

Trong ví dụ trên chúng ta đã lưu trữ mỗi Company trên những mảnh khác nhau, và mỗi Invoice được lưu trữ trên cùng mảnh với company của nó.

Chúng ta có thể thực hiện thao tác như Query, Load hay LuceneQuery và những mảnh thực sự sẽ được liên hệ để hoàn thành thao tác mà những thao tác này phụ thuộc vào việc hiện thực IShardResolutionStrategy.

|  |
| --- |
| *using (var session = documentStore.OpenSession())*  *{*  *//lấy tất cả dữ liệu từ mỗi mảnh một cách tự động*  *var allCompanies = session.Query<Company>()*  *.Customize(x => x.WaitForNonStaleResultsAsOfNow())*  *.Where(company => company.Region == "Asia")*  *.ToArray();*    *foreach (var company in allCompanies)*  *Console.WriteLine(company.Name);*  *}* |

Nếu chúng ta đang sử dụng DefaultShardResolutionStrategy trong trường hợp này thì nó chỉ gửi yêu cầu tới mảnh “Asia”.

##### Kết hợp giữa nhân bản và phân tán dữ liệu (Replication và Sharding)

Replication và Sharding là 2 tính năng mạnh mẽ của RavenDB. Chúng ta có thể sử dụng kết hợp Replication với Sharding.

Giả sử chúng ta chỉ lưu trữ dữ liệu Users trong RavenDB và chúng ta phân tán dữ liệu trên 2 nút dựa vào tên người dùng (người dùng tên bắt đầu là A-M trên một nút và N-Z trên nút thứ 2).

Phân tán với các nút chuyển đổi dự phòng chuyên dụng:

Trong thiết lập này, chúng ta có hai nút để lưu trữ thông tin người dùng, và hai nút phục vụ (slave) cho mỗi nút chính (master). Nếu một trong các nút chính bị hư hỏng, Raven tự động chuyển đổi sang các bản sao dự phòng.



Phân tán với nút dự phòng nội bộ:

Một lựa chọn khác là sử dụng sharding chủ yếu như một phương tiện giảm tải trên các máy chủ và thiết lập nhân bản giữa các nút khác nhau mà không có các nút chuyển đổi dự phòng chuyên dụng:



### 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 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 website: 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
* RavenDB.Embedded
* RavenDB.AspNetHost

Secondary Packages:

* RavenDB.Database
* RavenDB.Client.Debug
* RavenDB.Client.FSharp
* RavenDB.Client.MvcIntegration
* RavenDB.Server

Bundles Packages:

* RavenDB.Bundles.Authentication
* RavenDB.Bundles.Authorization
* RavenDB.Bundles.CascadeDelete
* RavenDB.Bundles.Expiration
* RavenDB.Bundles.IndexReplication
* RavenDB.Bundles.MoreLikeThis / Client
* RavenDB.Bundles.Replication
* RavenDB.Bundles.Quotas
* RavenDB.Bundles.UniqueConstraints / Client
* 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

# CHƯƠNG 5:XÂY DỰNG ỨNG DỤNG SỬ DỤNG RAVENDB

## Giới thiệu về ứng dụng

Để thực hành xây dựng ứng dụng sử dụng cơ sở dữ liệu NoSQL, mà cụ thể là cơ sở dữ liệu RavenDB, chúng tôi xây dựng một website cho phép các người dùng có thể thảo luận về vấn đề nào đó (với các chức năng cơ bản như Google Group). Website sử dụng công nghệ ASP.NET MVC 4 nhằm tận dụng các ưu điểm của mô hình phát triển web này.

## Lý do lựa chọn ứng dụng này

Để nhìn thấy được tốc độ của cơ sở dữ liệu NoSQL, khả năng làm việc với lượng dữ liệu cực kì lớn, khả năng phân tán dữ liệu… Website này là một lựa chọn phù hợp vì nó cũng yêu cầu đến các tính năng đó: hiệu suất cao, lưu trữ dữ liệu nhiều, phân tán dữ liệu và đặc biệt là không đòi hỏi khắc khe về tính nhất quán dữ liệu.

Website này cần lưu trữ một lượng lớn dữ liệu như: thông tin người dùng, thông tin nhóm, nội dung các bài đăng và các bình luận. Website yêu cầu đọc và ghi dữ liệu rất nhiều, tuy nhiên tần suất của việc đọc diễn ra nhiều hơn việc ghi nhiều lần. Đây là một thế mạnh của RavenDB so với các cơ sở dữ liệu NoSQL cùng loại. Với thiết kế đặc biệt của RavenDB và thiết kế dữ liệu dạng document sẽ giúp ứng dụng đạt được hiệu suất hoạt động cao hơn việc sử dụng cơ sở dữ liệu RDBMs.

Website chấp nhập việc dữ liệu không nhất quán trong một khoảng thời gian ngắn trước khi được cập nhập đúng lại. Các yêu cầu cập nhập chưa cần thiết sẽ được xử lý sau, các yêu cầu cân thiết để hiển thị, thông báo dữ liệu sẽ được ưu tiên trước. Như vây điều này cho phép ứng dụng chạy mượt mà hơn, giảm thời gian phải chờ đợi.

Ta vẫn có thể sử dụng cơ sở dữ liệu RDBMs để làm cơ sở dữ liệu cho website này. Tuy nhiên, chi phí sẽ đắt hơn do cần nhiều server hơn để có thể đạt hiệu suất ngang với NoSQL. Ngoài ra trong ứng dụng này, không cần thiết những tính chất đặc biệt của RDBMs như: khoá ngoại, mô hình dữ liệu quan hệ… Như vậy, với giải pháp sử dụng NoSQL cho ứng dụng này đã đáp ứng được hoàn toàn yêu cầu đề ra với một chi phí thấp nhất.

## Phân rã chức năng website

Website có 2 người dùng: Manager (quản lý) và Member (thành viên). Quyền Manager cao hơn quyền Member và được mô tả chi tiết như sơ đồ phân rã chức năng bên dưới:

 *Sơ đồ phân rã chức năng của Owner*

 *Sơ đồ phân rã chức năng của Manager*

*Sơ đồ phân rã chức năng của Member*

## Ý tưởng thiết kế

### Thiết kế mô hình 3 tầng (3 Tier): Clients – Web server – Database server

Database server

Server Asia

Server MiddelEast

Server American

Web server

View

Model

Controller

Clients Asia

Client n

Client 1

Client 2

Clients American

Client n

Client 1

Client 2

Clients MiddelEast

Client n

Client 1

Client 2

Các thành phần trong mô hình này:

**Clients**: các client nằm ở khắp nơi có thể truy cập vào website qua giao thức HTTP bằng web browser.

**Web server**: được thiết kế theo mô hình MVC, sử dụng ASP.NET MVC 4 framework. Tầng web server sẽ được mô tả chi tiết ở mục 5.4.2 Kiến trúc website.

**Database server**: database server sử dụng RavenDB. Sử dụng kĩ thuật Sharding để phân tán dữ liệu ra nhiều Server có thể đặt ở nhiều nơi. Tuỳ vào request đến từ đâu sẽ đi đến server thích hợp để lấy dữ liệu.

### Kiến trúc Website

Uses

Uses

Injects dependencies

Uses

Core

Domain

Interface

Service

Web

Implement

View

Controller

Model

UnityContainer

Uses

Các thành phần trong mô hình này:

**Project Core**: Chứa Domain và các Interface. Domain làm nhiệm vụ đại diện cho một đối tượng trong chương trình. Interface là các giao diện cung cấp các hàm giúp cho Controller làm việc, chủ yếu là lấy dữ liệu từ database.

**Project Server**: các lớp trong này hiện thực các Interface trong project Core. Các lớp này sẽ giao tiếp giữa Controller và Database.

**Project Web**: thiết kế theo mô hình MVC. Ngoài ra còn sử dụng UnityContainer để tiêm sự phụ thuộc (Inject Dependencies) cho các Interface trong project Core quan hệ với các class trong project Server. Các class trong project Web có thể sử dụng Domain để chứa đối tượng.

## Tổng quan ASP.NET MVC4

### Khái niệm

* Mẫu kiến trúc Model – View – Controller được sử dụng nhằm chia ứng dụng thành ba thành phần chính: model, view và controller. Nền tảng ASP.NET MVC giúp cho chúng ta có thể tạo được các ứng dụng web áp dụng mô hình MVC thay vì tạo ứng dụng theo mẫu ASP.NET Web Forms. Nền tảng ASP.NET MVC có đặc điểm nổi bật là nhẹ (lightweight), dễ kiểm thử phần giao diện (so với ứng dụng Web Forms), tích hợp các tính năng có sẵn của ASP.NET. Nền tảng ASP.NET MVC được định nghĩa trong namespace System.Web.Mvc và là một phần của name space System.Web.



* Vài nét về mô hình MVC:
  + **Models**: Các đối tượng Models là một phần của ứng dụng, các đối tượng này thiết lập logic của phần dữ liệu của ứng dụng. Thông thường, các đối tượng model lấy và lưu trạng thái của model trong CSDL. Ví dụ như, một đối tượng Product (sản phẩm) sẽ lấy dữ liệu từ CSDL, thao tác trên dữ liệu và sẽ cập nhật dữ liệu trở lại vào bảng Products ở SQL Server.

Trong các ứng dụng nhỏ, model thường là chỉ là một khái niệm nhằm phân biệt hơn là được cài đặt thực thụ, ví dụ, nếu ứng dụng chỉ đọc dữ liệu từ CSDL và gởi chúng đến view, ứng dụng không cần phải có tầng model và các lớp liên quan. Trong trường hợp này, dữ liệu được lấy như là một đối tượng model (hơn là tầng model).

* + **Views**: Views là các thành phần dùng để hiển thị giao diện người dùng (UI). Thông thường, view được tạo dựa vào thông tin dữ liệu model. Ví dụ như, view dùng để cập nhật bảng Products sẽ hiển thị các hộp văn bản, drop-down list, và các check box dựa trên trạng thái hiện tại của một đối tượng Product.
  + **Controllers**: Controller là các thành phần dùng để quản lý tương tác người dùng, làm việc với model và chọn view để hiển thị giao diện người dùng. Trong một ứng dụng MVC, view chỉ được dùng để hiển thị thông tin, controller chịu trách nhiệm quản lý và đáp trả nội dung người dùng nhập và tương tác với người dùng. Ví dụ, controller sẽ quản lý các dữ liệu người dùng gởi lên (query-string values) và gởi các giá trị đó đến model, model sẽ lấy dữ liệu từ CSDL nhờ vào các giá trị này.
* Ngày 15-08-2012, Microsoft đã cho ra phiên bản ASP.NET MVC 4 với khá nhiều tính năng mới, giao diện cũng được cải thiện khá nhiều so với phiên bản trước đó.

### Lịch sử phát triển.

|  |  |
| --- | --- |
| Ngày | Phiên bản |
| 10-12-2007 | ASP.NET MVC CTP |
| 13-03-2009 | ASP.NET MVC 1.0 |
| 10-03-2010 | ASP.NET MVC 2.0 |
| 13-01-2011 | ASP.NET MVC 3.0 |
| 20-09-2011 | ASP.NET MVC 4.0 Developer Preview |
| 15-02-2012 | ASP.NET MVC 4.0 Beta |
| 15-08-2012 | ASP.NET MVC 4 |

### Những tính năng mới

* ASP.NET Web API
* Enhancements to Default Project Templates:
* Mobile Project Template:
* Display Modes:
* jQuery Mobile, the View Switcher, and Browser Overriding:
* Task Support for Asynchronous Controllers:
* Azure SDK
* Database Migrations
* Empty Project Template
* Thêm Controller vào bất kỳ thư mục nào trong dự án
* Bundling và Minification
* Cho phép Logins từ Facebook và những tài khoản khác sử dụng OAuth và OpenID

## Phân tích, thiết kế hệ thống

### Use case

### Class diagram



*Class diagram cung cấp các chức năng chính cho website*

### Sequence diagram

#### Quản lý User



*Sequence diagram của chức năng Login*



*Sequence diagram thực hiện chức năng Register*

#### Quản lý Group



*Sequence diagram của chức năng Create Group*

**

*Sequence diagram của trang Home*



*Sequence diagram của chức năng Join Group*



*Sequence diagram của chức năng Accept Member*

#### Quản lý Topic



*Sequence diagram của chức năng Create Topic*



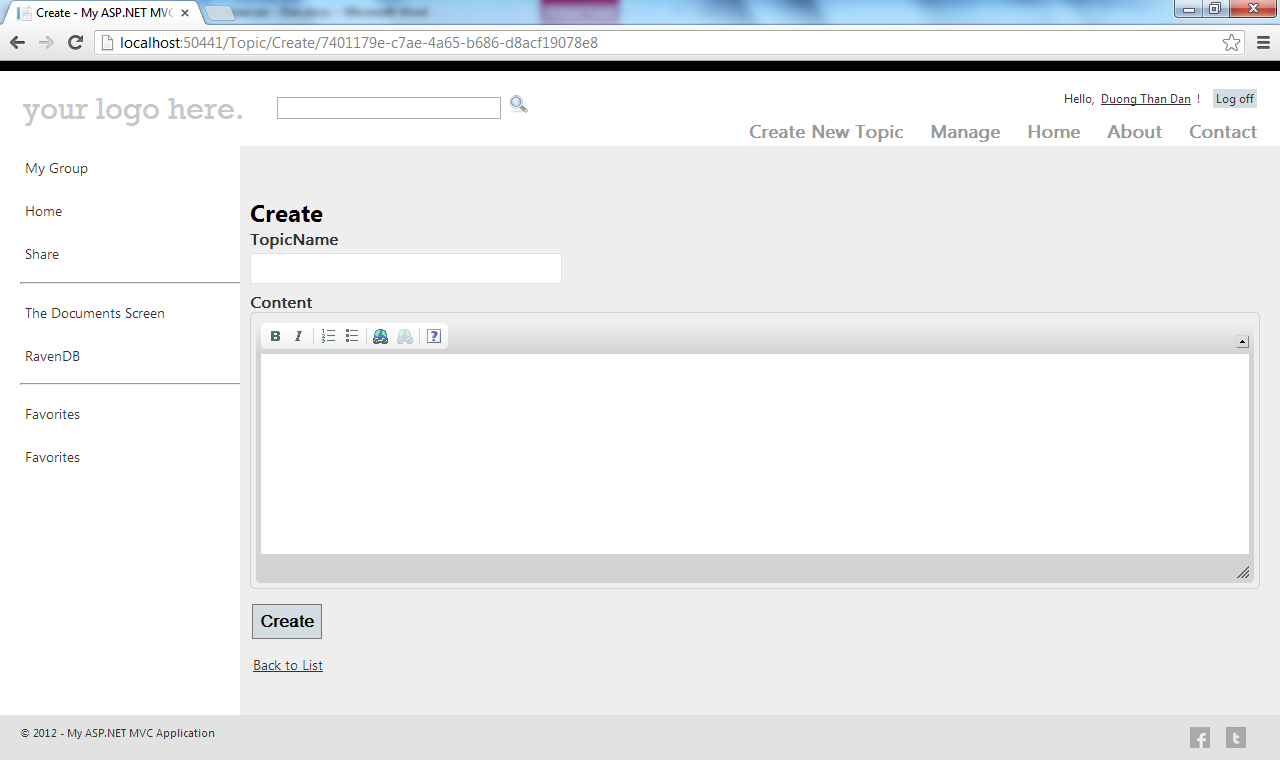
*Sequence diagram của chức năng đăng Comment*

#### Tìm kiếm

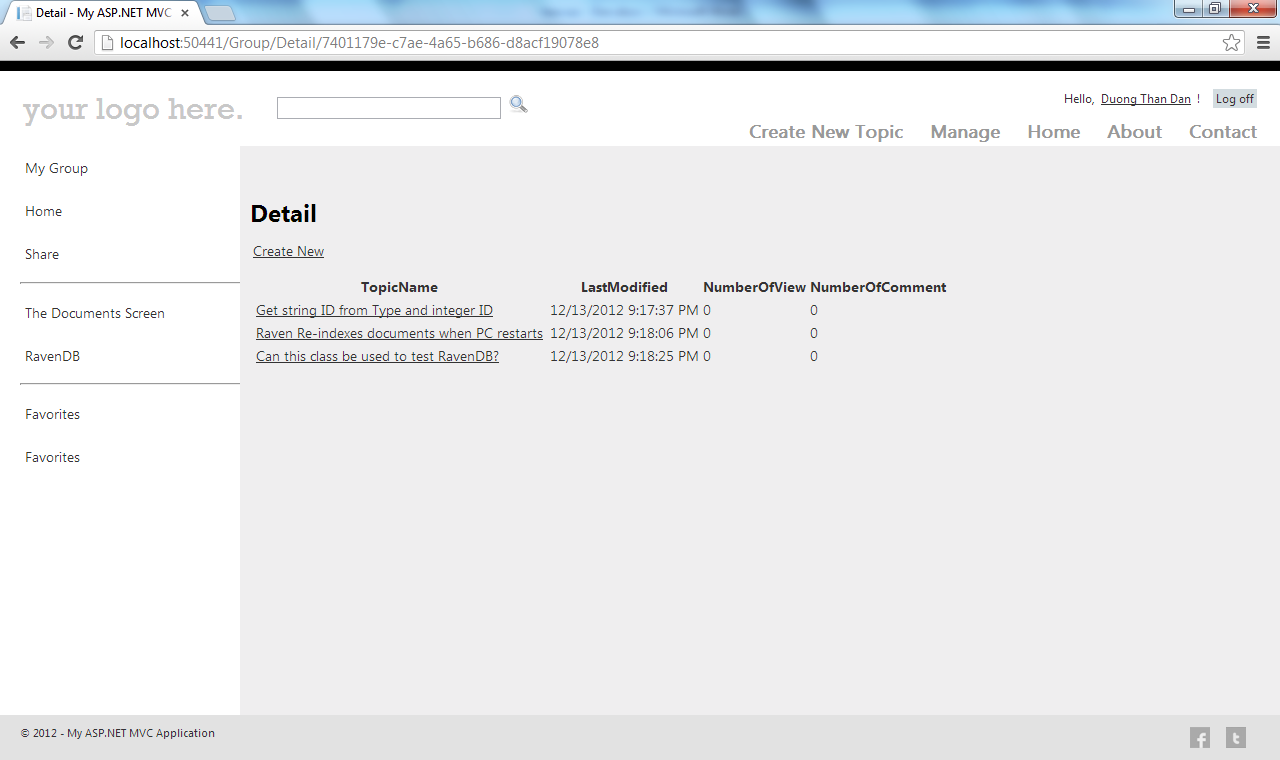


*Sequence diagram của chức năng tìm kiếm*

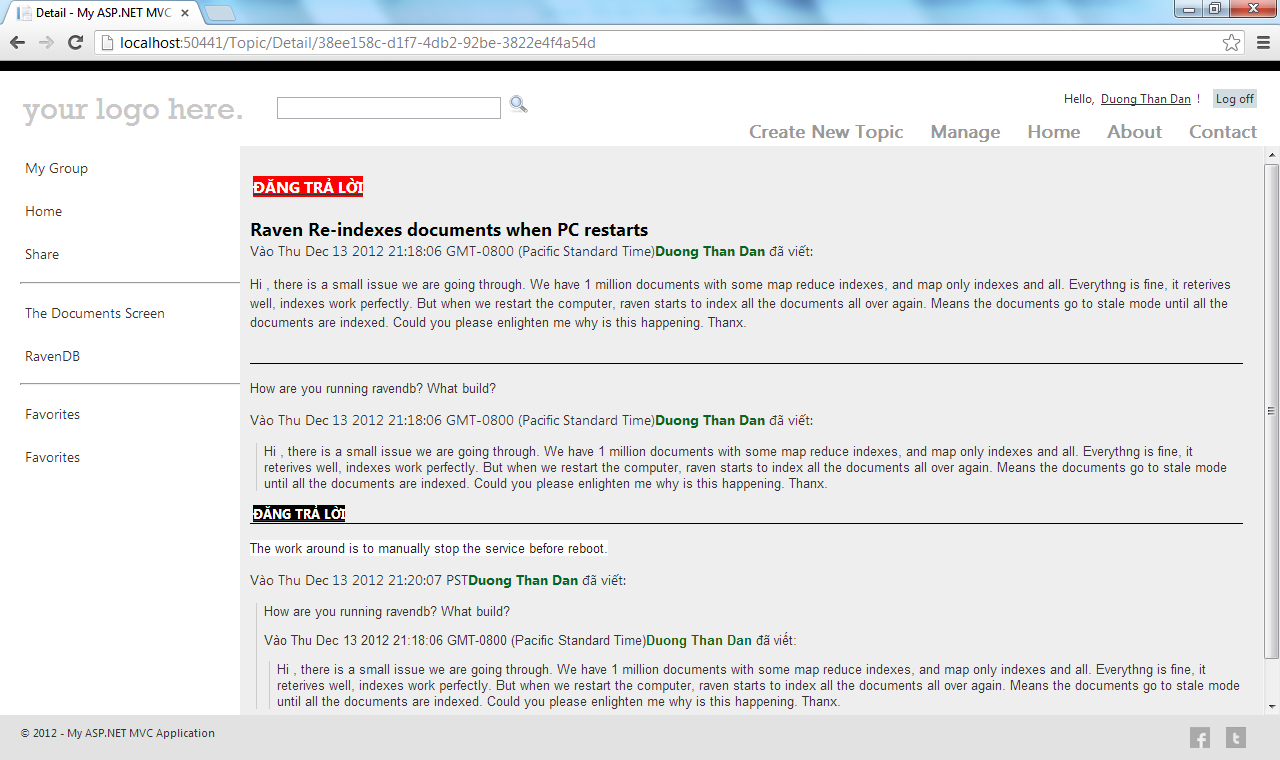
## Thiết kế giao diện



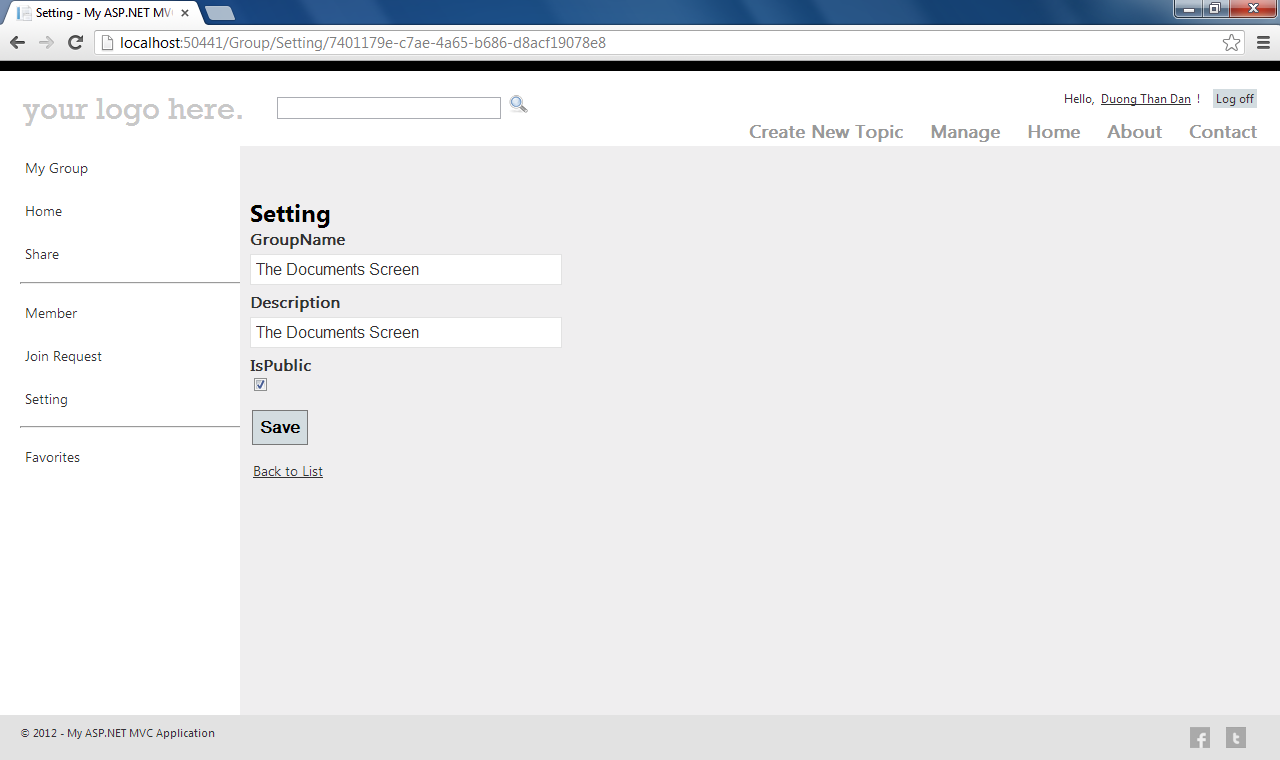
*Màn hình Create topic*



*Màn hình List topic*



*Màn hình Detail topic*



*Màn hình Setting Group*

## Test performance để thể hiện sức mạnh của NoSQL

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | SQL Server | RvenDB | **Độ lệch** | |
| TB  Lần | 00:00.083 | 00:00.322 | 00:00.238 | 286% |
| 1 | 00:00.311 | 00:00.952 | 00:00.641 |  |
| 2 | 00:00.026 | 00:00.582 | 00:00.556 |  |
| 3 | 00:00.027 | 00:00.023 | -00:00.004 |  |
| 4 | 00:00.027 | 00:00.026 | -00:00.001 |  |
| 5 | 00:00.026 | 00:00.025 | -00:00.001 |  |

*So sánh hiệu suất giữa SQL Server 2008 và RavenDB 960*

*(Làm việc với 20.000 record)*

# CHƯƠNG 6: TỔNG KẾT

## Kết quả đạt được

### Về mặt lý thuyết

* Tổng hợp và phân tích khá chi tiết về cơ sở dữ liệu NoSQL cùng với những ứng dụng thực tiễn của nó. Qua tài liệu này, người đọc có được cái nhìn bao quát về NoSQL và có thể ứng dụng nó vào các hệ thống cần lưu trữ rất nhiều dữ liệu.
* Trình bày và phân tích những loại khác nhau của NoSQL bao gồm: key-value store, column families, document database và graph database.
* Tìm hiểu về tính năng, đặc điểm và những lợi ích của một số loại document database phổ biến là: MongoDB, CouchDB, RavenDB.
* Đào sâu kiến thức về RavenDB và các triển khai một ứng dụng sử dụng RavenDB .

### Về mặt thực nghiệm

Xây dựng được một ứng dụng DaHu Groups (có các chức năng cơ bản giống Google Groups) sử dụng cơ sở dữ liệu RavenDB trên nền Web. Ứng dụng tuy không quá quy mô nhưng cũng đã áp dụng được những kĩ thuật cơ bản và nâng cao của RavenDB và ứng dụng đã thể hiện tốc độ vượt trội khi hoạt động với một lượng lớn dữ liệu, đáp ứng được yêu cầu đề ra.

## Hướng phát triển

* Tìm hiểu thêm vấn đề phân tán dữ liệu ở nhiều máy chủ.
* Tìm hiểu thêm các vấn đề quản lý transaction. Đặc biệt xử lý tình huống cơ sở dữ liệu phân tán trên nhiều máy chủ.
* Một số vấn đề như bảo mật, config server, backup và restore dữ liệu trên cơ sở dữ liệu NoSQL.

**DANH MỤC TÀI LIỆU THAM KHẢO**