

# Hàm băm phi mật mã (Non-cryptographic hash function)

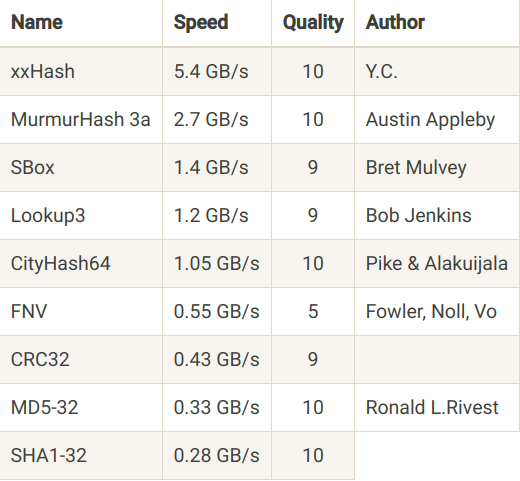
Ở phần trước chúng ta đẵ tìm hiểu về loại hàm băm mật mã, các bạn có thể xem tại [đây](https://medium.com/@tranndc/h%C3%A0m-b%C4%83m-v%C3%A0-%E1%BB%A9ng-d%E1%BB%A5ng-ph%E1%BA%A7n-1-e82e9866dbcb). Nếu chúng ta bỏ qua một số tính chất bảo mật nhất định để đổi lấy tốc độ thực thi nhưng vẫn giữ được độ phân phối tốt thì các hàm băm phi mật mã thích hợp hơn cả. Sự khác biệt giữa hàm băm phi mật mã và hàm băm mật mã đó chính độ khả thi tính toán để có thể tìm được thông điệp gốc từ dãy hash (reversable). Điều này có nghĩa là khả năng đụng độ (collision) sẽ cao hơn và khả năng lan truyền (avalanche effect) thấp hơn so với các` hàm băm mật mã dẫn đến các giá trị hash có thể trùng nhau. Một số hàm băm thuộc loại phi mật mã có thể kể đến như MurmurHash3, FNV1, SuperFastHash, CityHash, xxHash… Ở bài viết này tôi sẽ tập trung nói về hàm xxHash vì nó khá là mới mẻ và cực kỳ nhanh.

# xxHash

Ví dụ đơn cử ở team ZaloPay đang sử dụng hàm xxHash để băm các key. Mặc dù là một hàm băm rất nhanh nhưng độ bảo mật của xxHash vẫn đảm bảo tốt. xxHash không phải là một hàm băm mật mã mạnh, ví dụ như họ SHA nhưng nó vẫn đạt 10 điểm trên bài test SMHasher về độ phân phối (distribution), khả năng đụng độ (collision) và hiệu năng (performance). Số điểm này ngang với điểm của hai hàm băm mật mã trước kia là MD5 và SHA1. Tính duy nhất (Uniqueness) của xxHash là tương đương với MD5 nhưng khác biệt so với SHA1.

# Tốc độ

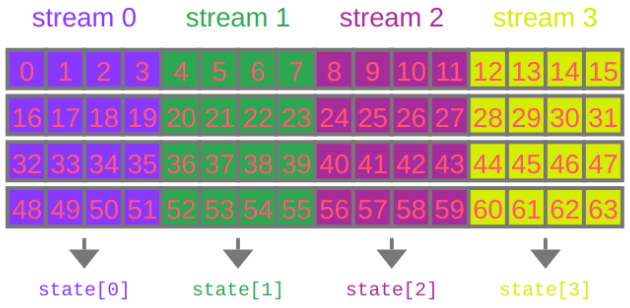
xxHash được thiết kế từ đầu để chạy nhanh nhất có thể trên các CPU hiện đại. Về tốc độ, ta tiến hành benchmark hai implementation của xxHash tương ứng với hai hệ kiến trúc 32-bit và 64-bit. Ở bản 32-bit, xxHash có thể hash lượng lớn data với tốc độ 5.4 Gb/s tức là 324 Gb/minute, nhanh gấp 16 lần so với MD5 (0.33 Gb/s) và 20 lần so với SHA1 (0.28 Gb/s). Ở phiên bản 64-bit, tốc độ được tăng lên rất nhiều: 13 Gb/s tức là khoảng 780 Gb/minute. Mặc dù ta có thể chạy phiên bản xxHash 64-bit trên kiến trúc 32-bit nhưng tốc độ lại bị giảm chỉ còn 1.9 Gb/s. Trong khi đó phiên bản xxHash 32-bit chạy trên kiến trúc 64-bit lại có tốc độ tăng lên thành 6.8 Gb/s. Bảng sau hiện thị thông số benchmark của các hàm băm trên bài test SMHasher biên dịch với Visual C trên Windows Seven 32-bit ở chế độ đơn luồng:



# Thuật toán

Hầu hết các hàm băm phi mật mã, như FNV duyệt qua từng byte của dữ liệu đầu vào. Điều này dẫn đến lệ thuộc khiến cho CPU bị đình trệ (stalling) vì làm việc trên byte hiện tại yêu cầu công việc xử lý ở byte trước đó phải hoàn tất. Có những thuật toán đã cải thiện vấn đề này đôi chút bằng cách duyệt qua từng khối byte một lần. Ví dụ như CRC Slicing-by-4/8/16 xử lý 4/8/16 byte mỗi bước thay vì chỉ 1 byte nên tốc độ hash được tăng. Tuy nhiên, chúng ta vẫn dính phải vấn đề ban đầu: làm việc trên block hiện tại yêu cầu công việc xử lý ở block trước đó phải hoàn tất!

Để dứt điểm triệt để vấn đề phụ thuộc giữa các thành phần của dữ liệu, xxHash chia nhỏ dữ liệu đầu vào thành bốn luồng độc lập. Mỗi luồng như vậy sẽ xử lý 4 byte mỗi bước và lưu vào state tạm thời. Chỉ duy nhất bước cuối cùng sẽ tổng hợp 4 state đã được sinh từ 4 luồng thành một state duy nhất. Lợi thế đáng kể của việc làm này đó chính là trình tạo mã có rất nhiều cơ hội để sắp xếp lại các opcodes để phòng tránh độ trễ.



# Locality Sensitive Hashing (LSH)

Bài toán “tìm hàng xóm gần nhất” (nearest neighbours) rất phổ biến khi áp dụng cho các tập dữ liệu lớn. Một số ứng dụng hay gặp đó là tìm bản sao hay các tài liệu giống nhau, tìm kiếm audio/video. Cách thông thường ta hay nghĩ đến đó là sử dụng kỹ thuật vét cạn (brute force) để kiểm tra tất cả các tổ hợp có thể để lấy được chính xác cặp hàng xóm gần nhau nhất. Tuy nhiên, cách làm đó rất khó scale cho tập dữ liệu lớn. Thế nên, các thuật toán xấp xỉ được đề cử cho vấn đề trên. Các thuật toán xấp xỉ sẽ không cho bạn câu trả lời chính xác nhưng chúng cung cấp độ xấp xỉ tốt với tốc độ nhanh và dễ scale.

Localilty Sensity Hashing (LSH) là một trong những thuật toán đó. Ý tưởng của LSH là sử dụng một hàm băm (có một họ các hàng băm cho LSH gọi là LSH families) để hash các điểm dữ liệu vào từng bucket sao cho các điểm dữ liệu gần nhau sẽ có xác suất cao rơi vào cùng một bucket, ngược lại chúng sẽ bị băm vào hai bucket khác nhau. Khi truy vấn, ta chỉ cấn tính mã băm tương ứng của dữ liệu đầu vào và trả về tất cả các điểm dữ liệu lưu trong bucket tương ứng với mã băm đó. Vì LSH mang tính chất xấp xỉ và phải cung cấp tốc độ thực thi cao nên các hàm băm sử dụng trong họ LSH thuộc các hàm băm phi mật mã.

LSH đã trở thành một kỹ thuật đa dụng để áp dụng vào những vấn đề như:

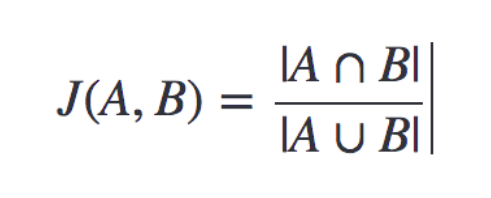
* Tìm trùng lặp gần nhất: LSH được dùng phổ biến cho việc phát hiện bản sao trong số lượng lớn các tài liệu, trang web và các tệp.
* Nghiên cứu bộ gen: các nhà sinh học thường sử dụng LSH để xác định các biểu hiện gen tương tự trong cơ sở dữ liệu bộ gen.
* Tìm kiếm hình ảnh quy mô lớn: Google sử dụng LSH với [PageRank](https://vi.wikipedia.org/wiki/PageRank#targetText=Pagerank%20l%C3%A0%20thu%E1%BA%ADt%20to%C3%A1n%20ph%C3%A2n,k%E1%BA%BFt%20trong%20t%E1%BA%ADp%20h%E1%BB%A3p%20%C4%91%C3%B3.) để xây dựng công nghệ tìm kiếm hình ảnh của họ là [VisualRank](https://en.wikipedia.org/wiki/VisualRank)
* Vân tay Audio/Video: Trong công nghệ truyền thông, LSH được sử dụng làm công nghệ vân tay cho các dữ liệu dạng Audio/Video.

Hãy cùng tìm hiểu thuật toán đằng sau LSH.

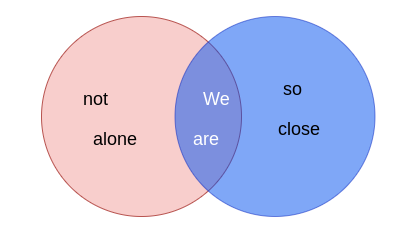
Ở bài viết này tôi sẽ lấy ví dụ về bài toán “tìm hai văn bản gần nhau nhất” để minh họa cách hoạt động của LSH. Giả sử ta có ba văn bản như sau:

A: "We are not alone"  
B: "We are so close"  
C: "Those are good old times"

Để phát hiện hai câu trên là trùng nhau hay không, dùng công thức tính khoảng cách giữa các string — Jaccard Index là một cách đơn giản và hiệu quả. Công thức tóm gọn như sau:

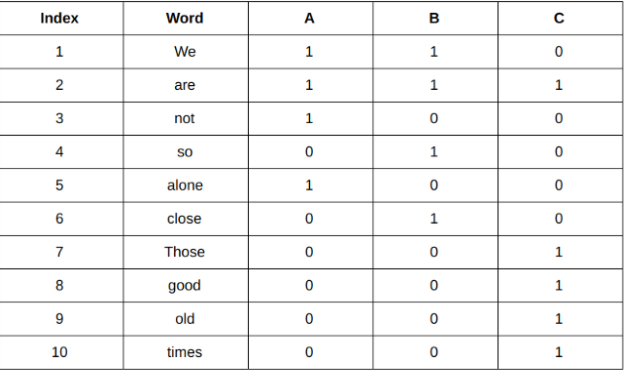


Để áp dụng công thức trên dễ dàng, ta sẽ chia hai câu trên thành từng từ. Minh họa Jaccard Index cho hai câu A và B như sau:



Ta thấy số lượng từ giao nhau là 2 và tổng số lượng từ là 6 nên Jaccard Index sẽ có giá trị 2/6 = 0.333.

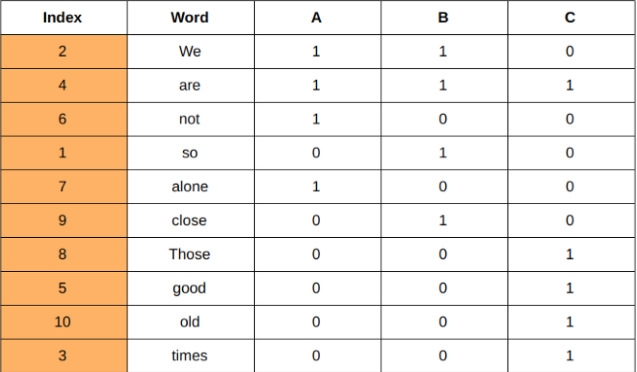
Mọi việc tưởng như thật đơn giản nhưng cách làm trên lại không dễ scale: số lượng các phép so sánh là 0.5 \* n(n -1) nên độ phức tạp thời gian sẽ là O(n²). Ta có thể biểu diễn lại hai câu trên dưới dạng ma trận Boolean để thuận tiện cho việc so sánh. Ta xây dựng một tập tất cả các từ riêng biệt tổng hợp từ hai câu, nếu một từ nằm trong một câu thì ta đánh dấu là 1, ngược lại là 0:



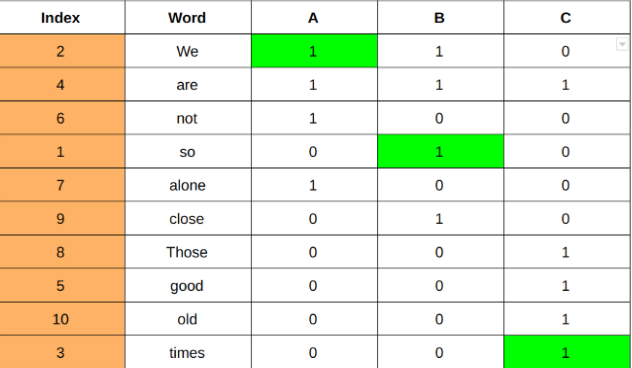
Đến đây vẫn chưa tối ưu vì lưu trữ một ma trận như vậy rất tốn bộ nhớ. Có một cách để giải quyết vấn đề này đó chính là dùng hàm băm. Đây cũng là phần quan trọng nhất trong bài viết về LSH. Có nhiều hàm băm cho ta lựa chọn nhưng hàm băm thích hợp cho Jaccard Index đó chính là **MinHashing**.

Các bước quan trọng trong quá trình này bao gồm:

**Bước 1:** Hoán vị ngẫu nhiên index dòng của ma trận Boolean ta đã xây dựng:



**Bước 2:** Hàm hash sử dụng trong MinHashing là: ở mỗi cột tương ứng với các văn bản, duyệt đến khi ta tìm được một ô có giá trị 1 mà tại đó đối chiếu qua cột index (đã được hoán vị ngẫu nhiên) có giá trị nhỏ nhất. Giá trị hash chính là giá trị của index đó.

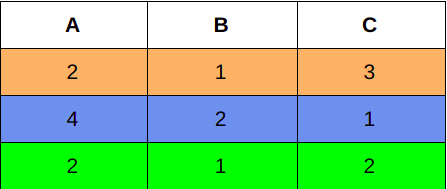


Như hình trên ta hash ra được một dòng trong ma trận chữ ký là [2, 1, 3]

Làm lại các bước trên nhiều lần với mỗi lần cột index được hoán vị ngẫu nhiên ta được một ma trận chữ ký đầy đủ hơn:



Ma trận chữ ký hash ra được sẽ là:



Đến đây ta đã có thể tính được độ giống nhau dựa trên ma trận chữ ký: Sự giống nhau giữa các chữ ký là tỉ lệ mà số hàm min-hash (các hàng) có cùng giá trị với nhau. Ví dụ với ma trận chữ ký ở trên, sự giống nhau giữa A và B là 2/3 vì chúng có cột 1 và 3 giống nhau.