Kỷ yếu Hội nghị Khoa học Quốc gia lần thứ IX "Nghiên cứu cơ bản và ứng dụng Công nghệ thông tin (FAIR'9)"; Cần Thơ, ngày 4-5/8/2016
DOI: 10.15625/vap.2016.00091

TỐI ƯU HÓA JOIN ĐỆ QUY TRÊN TẬP DỮ LIỆU LỚN TRONG MÔI TRƯỜNG SPARK

Phan Thượng Cang[1], Trần Thị Tố Quyên[1], Phan Anh Cang[2]

[1] Khoa Công nghệ thông tin và Truyền thông, Đại học Cần Thơ
[2] Khoa Công nghệ thông tin, Trường Đại học Sư phạm Kỹ thuật Vĩnh Long
ptcang@cit.ctu.edu.vn, tranthitoquyen@cit.ctu.edu.vn, cangpa@vlute.edu.vn

TÓM TẮT— *MapReduce đã trở thành một mô hình lập trình chính cho phân tích và xử lý dữ liệu lớn trong những năm gần đây. Tuy nhiên, mô hình này vẫn còn tồn tại một số mặt hạn chế như chưa hỗ trợ đầy đủ cho các tính toán lặp, cơ chế bộ nhớ đệm (cache), và các hoạt động với đa đầu vào (multiple inputs). Ngoài ra, các chi phí cho việc đọc/viết và truyền thông dữ liệu của mô hình còn quá tốn kém. Một trong những hoạt động phức tạp đáng chú ý và thường được sử dụng trong MapReduce đó là Join đệ quy. Nó đòi hỏi những đặc trưng xử lý mà cũng chính là những hạn chế của MapReduce. Vì vậy, trong nghiên cứu này, chúng tôi đề xuất một số giải pháp hiệu quả cho xử lý Join đệ quy trên nền tảng xử lý dữ liệu lớn thế hệ mới Spark. Đề xuất của chúng tôi đã loại bỏ một lượng lớn dữ liệu dư thừa được tạo ra trong các xử lý lặp của Join đệ quy, tận dụng những lợi thế của việc xử lý trong bộ nhớ và cơ chế bộ nhớ đệm để giảm thiểu các chi phí có liên quan. Thông qua mô hình chi phí và các thực nghiệm, nghiên cứu này chỉ ra rằng các giải pháp của chúng tôi đã cải tiến đáng kể hiệu suất thực thi của Join đệ quy trong môi trường MapReduce.*

Từ khóa— *Big data analytics, recusrsive join, map reduce, spark.*

I. GIỚI THIỆU

Trong thời đại bùng nổ thông tin như hiện nay, thuật ngữ "Big Data" dần trở nên quen thuộc và đặt ra nhiều thách thức trong các nghiên cứu như công nghệ tìm kiếm (search-engines), phân tích mạng xã hội (social network analysis), phân tích dữ liệu Web (Web-data analysis), phân tích giám sát mạng (network-monitoring analysis), các mô phỏng lớn (massive-scale simulations), cảm biến thông lượng cao (high-throughput sensors), v.v.

Để xử lý lượng dữ liệu cực lớn cho các công việc trên, chúng ta cần có những mô hình lập trình phân tán mới chạy trên các cụm máy tính (clusters). Ý tưởng chính của việc tính toán phân tán là chia bài toán thành những bài toán con và giải quyết trên các máy riêng biệt nhau được kết nối trong một cụm máy tính. MapReduce [1] được Google đề xuất năm 2004 đã trở thành mô hình chuẩn và phổ biến nhất hiện nay để xử lý dữ liệu lớn trên các hệ thống song song và phân tán. Một cụm máy tính thực thi công việc MapReduce có thể gồm hàng ngàn nút tính toán (computing nodes) với khả năng chịu lỗi cao, thích hợp với những ứng dụng xử lý dữ liệu cực lớn, song song và co giãn. Mô hình MapReduce tương thích với nhiều loại giải thuật như phân tích tài liệu web lớn (web-scale document analysis) [1], thực thi câu truy vấn quan hệ (relational query evaluation) [2] và xử lý ảnh quy mô lớn (large scale image processing) [3]. Tuy nhiên, nó không được thiết kế cho các hoạt động với đa đầu vào (multiple inputs). Ngoài ra, nó cũng không hỗ trợ tốt cho nhiều ứng dụng xử lý dữ liệu lớn đòi hỏi sự tính toán lặp lại như PageRank [4], HITS (HypertextInduced Topic Search) [5], các câu truy vấn đệ quy (recursive relational queries) [6], phân cụm dữ liệu (clustering) [7], phân tích mạng neutron (neural network analysis) [8], phân tích mạng xã hội (social network analysis) [9] và phân tích lưu lượng dữ liệu internet (Internet traffic analysis) [10]. Những ứng dụng này liên quan đến các tính toán lặp đi lặp lại liên tục trên các tập dữ liệu lớn cho đến khi chúng đạt đến một điều kiện dừng hay một điểm dừng (a fix point). Để khắc phục những hạn chế đó, các lập trình viên phải cài đặt giải thuật lặp lại trong môi trường MapReduce bằng cách thực thi nhiều chuỗi công việc và tự quản lý công việc lặp. Điều này dẫn đến việc đọc và viết dữ liệu phải thực hiện lại nhiều lần, làm tăng đáng kể các chi phí như I/O, CPU, và truyền thông, cũng như ảnh hưởng rất nhiều đến tốc độ xử lý của ứng dụng. Đây chính là những thách thức không nhỏ đối với việc xử lý dữ liệu lớn trong môi trường MapReduce.

Join [11, 12] là một phép kết nối từ hai hoặc nhiều tập dữ liệu trong một cơ sở dữ liệu. Join thường được sử dụng trong các câu truy vấn dữ liệu tiêu biểu với chi phí và độ phức tạp lớn. Các dạng Join có thể là Join hai chiều (two-way join), Join đa chiều (multi-way join) [13], Join chuỗi (chain join) [14] và Join đệ quy (recursive join) [15–17]. Các truy vấn Join trên các tập dữ liệu càng trở nên phức tạp trong ngữ cảnh Big Data. Trong phạm vi nghiên cứu này, chúng tôi tập trung nghiên cứu xử lý Join đệ quy, một dạng Join phức tạp và có chi phí thực thi rất lớn được áp dụng trong nhiều lĩnh vực như PageRank, khai thác dữ liệu đồ thị (graph mining), giám sát mạng máy tính, mạng xã hội, tin sinh học (bioinformatics), v.v.

Một ví dụ điển hình cho ứng dụng Join đệ quy là câu truy vấn khám phá các mối quan hệ của một cá nhân trong một mạng xã hội được định nghĩa như sau:

$$\text{Friend}(x, y) \leftarrow \text{Know}(x, y);$$

$$\text{Friend}(x, y) \leftarrow \text{Friend}(x, z) \bowtie \text{Know}(z, y);$$

Một cá nhân *x* là bạn của *y* nếu *x* quen biết trực tiếp với *y*. Một cá nhân *x* cũng sẽ là bạn của *y* nếu *x* là bạn của *z* và *z* quen biết với *y*. Điều này cũng đồng nghĩa với câu truy vấn tìm tất cả bạn của các bạn của một cá nhân (friends of friends of a user).

Việc thực thi một câu truy vấn Join đệ quy thông thường sẽ bao gồm hai pha công việc được lặp lại. Thứ nhất, *pha xử lý Join*: đọc, xử lý và vận chuyển một hoặc nhiều tập dữ liệu đầu vào; loại bỏ các dòng không thoả mãn điều kiện join; kết hợp các dòng thoả mãn điều kiện join để tạo ra kết quả tạm thời. Thứ hai, *pha xác định tập dữ liệu tăng cường* (incremental dataset): đọc lại tất cả dữ liệu kết quả đã sinh ra từ những vòng lặp trước để loại bỏ những kết quả bị trùng lắp trong tập kết quả tạm thời; ghi lại tập kết quả không trùng với các tập kết quả trước đó (còn gọi là tập dữ liệu tăng cường). Tập dữ liệu tăng cường này sẽ là đầu vào cho pha xử lý Join kế tiếp. Hai pha công việc này sẽ được thực hiện lặp lại cho đến khi không phát hiện ra kết quả mới nào.

Với những hạn chế đã được chỉ ra ở trên thì rõ ràng MapReduce không hỗ trợ trực tiếp cho các hoạt động như Join đặc biệt là Join đệ quy. Do đó, việc thực thi câu truy vấn Join đệ quy trên tập dữ liệu lớn trong môi trường MapReduce trở thành mối quan tâm hàng đầu và cũng chứa đựng những thách thức lớn cho các nhà nghiên cứu. Chính vì vậy, nhóm nghiên cứu chúng tôi tập trung vào việc tìm kiếm và đề xuất các giải pháp tối ưu cho xử lý Join đệ quy. Để thực hiện được điều đó, nhóm chúng tôi trước tiên tiến hành nghiên cứu các thành phần mở rộng trên nền tảng MapReduce để hỗ trợ hiệu quả cho việc thực thi các công việc lặp lại nhiều lần và cơ chế bộ nhớ đệm (cache) mà nó có thể giữ lại tập dữ liệu không đổi trong xử lý lặp. Sau đó, chúng tôi sẽ đưa ra các phương thức để loại bỏ những phần tử dư thừa không tham gia vào hoạt động Join.

II. CÁC NGHIÊN CỨU LIÊN QUAN

A. Join đệ quy trong môi trường MapReduce

Câu truy vấn Join đệ quy của một quan hệ cũng được xem như là một câu truy vấn bao đóng bắc cầu của quan hệ đó [17]. Trên thực tế, đã có rất nhiều thuật toán được thiết kế để tính bao đóng bắc cầu của một quan hệ trong cơ sở dữ liệu truyền thống như Naive [24], Semi-naive [1–3], Smart [27, 28], Minimal evaluations [28], Warshall [30] và Warren [31]. Chúng được phân loại thành hai nhóm chính, các thuật toán lặp (Naive, Semi-naive, Smart, Minimal evaluations) và các thuật toán tính trực tiếp (Warshall và Warren). Tuy nhiên, những thuật toán này không phải lúc nào cũng phù hợp để thực hiện trong môi trường MapReduce.

Một số nghiên cứu gần đây đã tìm thấy giải pháp cho vấn đề này. Afrati et al. [32, 33] đã đề xuất việc thực thi đệ quy trên một cụm máy tính (cluster) để tính bao đóng bắc cầu cho câu truy vấn đệ quy. Các tác giả đã chỉ ra giải pháp để làm giảm đáng kể số lần lặp cần thiết cho bao đóng bắc cầu phi tuyến (nonlinear transitive closures). Giải pháp sử dụng hai nhóm tác vụ gồm nhóm tác vụ Join (Join tasks) và nhóm tác vụ loại bỏ sự trùng lắp (Dup-elim tasks). Nhóm tác vụ Join sẽ thực hiện việc tính toán join các bộ dữ liệu (còn gọi là tuples). Nhóm tác vụ còn lại sẽ loại bỏ các bộ dữ liệu kết quả trùng lắp trước khi phân phát đến nhóm tác vụ Join. Kết quả là số lần lặp cho thực thi Join đệ quy đã giảm đến $O(log_2 n)$ thay vì $O(n)$ trên một đồ thị n-node. Một trở ngại lớn của giải pháp này là các tác vụ (tasks) chạy đệ quy trong thời gian dài làm tăng nguy cơ thất bại. Ngoài ra, giải pháp phải sửa đổi một số đặc trưng của mô hình MapReduce như cơ chế phục hồi lỗi và thuộc tính khóa các tác vụ Map (blocking property). Bên cạnh đó, nó được sử dụng để tính bao đóng bắc cầu phi tuyến và chi phí truyền thông của nó thường là lớn hơn nhiều so với bao đóng bắc cầu tuyến tính (linear transitive closures) do sự sao chép kết quả đầu ra của các tác vụ loại bỏ sự trùng lắp.

Pregel [34] thực hiện sự đệ quy thực sự trên một đồ thị bằng cách sử dụng mô hình BSP (Bulk Synchronous Parallel) và trong mỗi khoảng thời gian nó sẽ kiểm tra tất cả các tác vụ (được gọi là các checkpoint). Nếu có một tác vụ thất bại thì tất cả các tác vụ sẽ phải thực thi lại tại các checkpoint trước đó. Điều này là một hạn chế lớn của giải pháp.

Hadoop [35] cung cấp một nền tảng xử lý dữ liệu lớn theo mô hình MapReduce và một hệ thống tập tin phân tán HDFS (Hadoop Distributed File System). Nó chạy trên trên các cụm máy tính phân tán mà vẫn đảm bảo độ tin cậy và khả năng chịu đựng lỗi. Tuy nhiên, do phát triển trên ý tưởng ban đầu của MapReduce nên Hadoop không hỗ trợ tốt các hoạt động có đa đầu vào (multiple inputs) và xử lý lặp như Join đệ quy.

HaLoop [36] là phiên bản chỉnh sửa của Hadoop, được thiết kế nhằm hỗ trợ hiệu quả cho các ứng dụng mô hình MapReduce cần cơ chế bộ nhớ đệm và xử lý lặp như Join đệ quy. HaLoop sử dụng hệ thống phân tán để lưu trữ dữ liệu đầu vào và đầu ra của các công việc MapReduce. Nó thực hiện sự đệ quy bằng cách lặp lại công việc MapReduce và giảm thiểu truyền thông của bộ nhớ đệm đầu vào của Mapper và đầu vào/ra của Reducer. Giải pháp này có thể tránh việc đọc lại và truyền lại dữ liệu qua mạng, tất nhiên nó vẫn phải đọc lại bộ nhớ đệm. Một hạn chế là các tác vụ nên hoạt động trong các vòng lặp đồng bộ và đầu ra của một tác vụ phải được gửi đến một Map/Reduce kế tiếp. Ngoài ra, một nhược điểm của việc thực hiện bộ nhớ đệm trong HaLoop hiện nay xuất phát từ hoạt động viết lại hoàn toàn bộ nhớ đệm cho mỗi lần lặp. Về mặt ý tưởng, HaLoop hỗ trợ tốt để xử lý truy vấn đệ quy trên các tập dữ liệu lớn. Tuy nhiên, HaLoop chỉ dừng lại ở mức nghiên cứu. Trên các thực nghiệm của chúng tôi [37], phương thức cache của HaLoop còn phát sinh những lỗi không kiểm soát khi cache trên đĩa cứng, không cache được tập dữ liệu tăng cường qua mỗi vòng lặp. Hiện nay HaLoop không còn được phát triển và hỗ trợ nữa.

Gần đây nhất, Shaw et al. [38] đã đề xuất một giải pháp tối ưu cho Join đệ quy sử dụng giải thuật Semi-Naïve trong môi trường MapReduce của HaLoop. Giải pháp sẽ thực thi việc lặp lại trên hai công việc MapReduce bao gồm công việc Join và công việc tính tập dữ liệu tăng cường. Giải pháp sử dụng bộ nhớ đệm cho đầu vào các Reducer (Reducer Input Cache) của HaLoop để giảm thiểu các chi phí có liên quan. Tuy nhiên, giải pháp này vẫn không tránh khỏi những hạn chế của HaLoop. Bên cạnh đó, các bộ dữ liệu được tạo ra bởi công việc Join phải được đọc lại và được chuyển qua mạng lần nữa trong công việc tính tập dữ liệu tăng cường. Hơn thế nữa, chi phí đọc viết bộ nhớ đệm là

đáng kể do tất cả các tập dữ liệu tăng cường của các lần lặp trước đó phải được viết, sắp chỉ mục và đọc lại để dò tìm sự trùng lắp bởi công việc tính tập dữ liệu tăng cường. Điều đáng lưu ý là cả bộ nhớ đệm cũng phải được viết lại mỗi khi có một bộ mới trong tập dữ liệu tăng cường được tìm thấy. Ngoài ra, dữ liệu không tham gia vào công việc Join vẫn được xử lý và truyền qua mạng mà không bị loại bỏ. Tất cả điều này dẫn đến chi phí thực thi Join đệ quy là quá lớn và chưa hiệu quả.

Từ những hạn chế của các giải pháp đã nêu, chúng ta cần tìm kiếm một nền tảng xử lý hỗ trợ tốt hơn cho cơ chế bộ nhớ đệm và thực thi lặp các công việc MapReduce. Trên nền tảng xử lý dữ liệu lớn đó, chúng ta sẽ tiến hành tối ưu hóa Join đệ quy một cách có hiệu quả. Việc loại bỏ những dữ liệu không tham gia vào Join cũng cần được xem xét một cách thấu đáo.

B. Apache Spark

Spark [18] là một nền tảng được viết bằng ngôn ngữ Scala, cho phép xử lý dữ liệu lớn phân tán một cách hiệu quả và nhanh chóng. Spark tương thích với nhiều hệ thống lưu trữ tập tin như HDFS, Cassandra [19], HBase [20] và Amazon S3 [21]. Spark có tốc độ xử lý nhanh gấp 100 lần so với Hadoop MapReduce khi được cache trên bộ nhớ, hoặc nhanh hơn gấp 10 lần nếu được cache trên đĩa [22].

Spark hỗ trợ các ngôn ngữ lập trình như Scala, Python, Java, và cung cấp nhiều công cụ lập trình cấp cao. Đặc điểm nổi bật của Spark là các tập dữ liệu phân tán có khả năng phục hồi RDD (Resilient Distributed Dataset), một kiểu dữ liệu tập hợp phân tán (distributed collection) có thể lưu tạm thời trên bộ nhớ RAM, có khả năng chịu lỗi cao và tính toán song song. RDDs hỗ trợ 2 kiểu hoạt động: *transformations* và *actions*. Transformations là một thao tác "*lazy*", có nghĩa là thao tác này sẽ không thực hiện ngay lập tức, mà chỉ ghi nhớ các bước thực hiện lại. Thao tác này chỉ được thực hiện khi trong quá trình thực hiện có Actions được gọi thì công việc tính toán của Transformations mới được diễn ra. Mặc định, mỗi RDD sẽ được tính toán lại nếu có Actions gọi nó. Tuy nhiên, RDDs cũng có thể được lưu lại trên bộ nhớ RAM bằng lệnh persist (hoặc cache) để sử dụng sau này. Actions sẽ trả về kết quả cho chương trình điều khiển (driver) sau khi thực hiện hàng loạt các tính toán trên các RDD.

1. Khả năng xử lý lặp của Spark

Spark là một công cụ hỗ trợ mạnh mẽ cho xử lý lặp trên các tập dữ liệu lớn. Một trong những phương thức của Spark thích hợp nhất cho loại xử lý này là thông qua RDD. RDD được mô tả như là "..., *cấu trúc dữ liệu song song chịu lỗi cho phép người dùng giữ lại kết quả trung gian trong bộ nhớ, kiểm soát phân vùng của chúng để tối ưu hóa vị trí lưu trữ dữ liệu, và xử lý chúng thông qua một tập các phép toán*." [23]. Bằng cách sử dụng RDD, một lập trình viên có thể "*ghim*" (pin) các tập dữ liệu lớn của họ vào bộ nhớ. Điều này giúp cho Spark RDD hỗ trợ xử lý lặp hiệu quả hơn so với việc đọc một lượng lớn dữ liệu từ đĩa cho mỗi lần lặp như Hadoop MapReduce.

Việc xử lý lặp của Hadoop MapReduce được thực hiện như một chuỗi các công việc nối tiếp nhau mà ở đó các kết quả trung gian phải viết đến HDFS và sau đó chúng phải được đọc lại để làm đầu vào cho công việc kế tiếp. Trong khi đó, Spark sẽ đọc dữ liệu đầu vào từ HDFS, thực hiện một loạt các hoạt động lặp đi lặp lại đối với dữ liệu dạng RDD, và sau cùng mới viết đến HDFS.

Rõ ràng, Spark RDD chạy nhanh hơn Hadoop MapReduce bởi vì ở mỗi tác vụ dữ liệu được nạp lên bộ nhớ và xử lý ở đó, những tác vụ sau đó có thể sử dụng lại dữ liệu nằm sẵn trên bộ nhớ cục bộ thay vì phải đọc ghi liên tục vào HDFS như Hadoop MapReduce.

Tương tự, các thuật toán xử lý đồ thị như PageRank đòi hỏi sự lặp lại nhiều lần trên cùng một tập dữ liệu và một cơ chế truyền thông điệp, do vậy cần một chương trình MapReduce. Tuy nhiên, cơ chế của MapReduce hoạt động liên quan đến đọc/ghi trên HDFS quá nhiều. Điều này không hiệu quả vì nó không những liên quan đến việc đọc và ghi dữ liệu vào đĩa mà còn liên quan đến hoạt động nhập/xuất và sao chép dữ liệu trên cụm máy tính với khả năng chịu lỗi. Ngoài ra, mỗi lần lặp MapReduce có độ trễ rất cao, và các lần lặp kế tiếp chỉ có thể bắt đầu khi các công việc trước đây đã hoàn toàn kết thúc. Spark chứa một thư viện đồ thị tính toán được gọi là GraphX. Kết hợp khả năng tính toán trong bộ nhớ và hỗ trợ khả năng xử lý đồ thị của Spark góp phần cải thiện hiệu suất của thuật toán so với chương trình MapReduce truyền thống. Ngoài ra, phần lớn các thuật toán máy học cũng đòi hỏi việc thực thi các thuật toán có tính lặp lại. Các thuật toán này liên quan đến hiện tượng nghẽn cổ chai I/O trong việc triển khai MapReduce. MapReduce sử dụng các tác vụ song song tạo ra gánh nặng cho các giải thuật đệ quy. Spark có một thư viện máy học gọi là MLib mà nó bao gồm các giải thuật mức cao phù hợp với sự lặp lại và hiệu quả hơn so với sử dụng MapReduce của Hadoop.

2. Cơ chế bộ nhớ đệm của Spark

Do RDD sẽ khởi tạo lại mỗi lần thực hiện một Action nên sẽ tốn rất nhiều thời gian nếu ta gặp phải trường hợp một RDD được sử dụng lại nhiều lần, chi phí cho công việc này là rất lớn. Vì thế, Spark hỗ trợ một cơ chế gọi là persist hay cache. Khi chúng ta yêu cầu Spark thực hiện persist một RDD, những nút chứa RDD đó sẽ lưu những RDD này vào bộ nhớ, và nút đó sẽ chỉ tính toán một lần. Nếu việc persist thất bại, Spark sẽ tự tính lại những phần bị thiếu nếu cần thiết.

C. Bộ lọc Bloom giao

1. Bộ lọc Bloom

Bộ lọc Bloom (Bloom Filter, BF) [39] được giới thiệu năm 1970 bởi Burton Bloom là một cấu trúc dữ liệu xác suất được sử dụng để kiểm tra xem một phần tử có nằm trong một tập hợp hay không.

Hình 1. Cấu trúc bộ lọc Bloom

Hình 1 biểu diễn cấu trúc bộ lọc Bloom gồm *m* bit, *k* hàm băm độc lập và một tập hợp *S* gồm *n* phần tử được biểu diễn bởi BF(*S*). Hoạt động của bộ lọc BF(*S*) được mô tả như sau:

- Khởi đầu, các bit của BF được thiết lập bằng 0.
- Khi thêm một phần tử x thuộc *S* vào bộ lọc, *k* vị trí của *x* trong mảng bit BF được xác định bởi *k* hàm băm và các vị trí này sẽ được đặt với giá trị là 1. Thực hiện công việc trên với tất cả phần tử của *S* để có được BF biểu diễn cho tập *S* hay còn gọi là BF(*S*).
- Để kiểm tra một phần tử *z* có thuộc *S* hay không, chúng ta cần kiểm tra tất cả *k* vị trí của *z* (tương ứng *k* hàm băm trên giá trị *z*):
  - Nếu tất cả các giá trị tại các vị trí đó đều là 1 thì có thể $z \in S$.
  - Ngược lại, nếu tồn tại ít nhất một trong các vị trí này có giá trị bằng 0 thì chắc chắn rằng $z \notin S$.

Trong một số trường hợp, một hàm băm cho nhiều phần tử có thể trả về cùng một giá trị vì vậy một phần tử không tồn tại trong *S* cũng có thể có giá trị băm tại vị trí đó bằng 1. Chính vì lý do này mà BF có thể trả về những phần tử dương tính giả (false positives), nhưng nó không bao giờ trả về phần tử âm tính giả (false negatives). Phần tử dương tính giả là phần tử được BF xác định là thuộc tập *S* nhưng thực ra nó không thuộc *S*. Phần tử âm tính giả là phần tử được BF xác định là không thuộc tập *S* nhưng thực ra nó lại thuộc *S*.
Bộ lọc Bloom có nhiều ưu điểm như sau:

- Không gian lưu trữ hiệu quả (space eciency): Kích thước của bộ lọc là cố định, không phụ thuộc vào số lượng phần tử *n* nhưng nó có mối liên hệ giữa mảng bit *m* và dương tính giả theo xác suất [40]:

$$f = (1-p)^k = \left(1-\left(1-\frac{1}{m}\right)^{kn}\right)^k \approx \left(1-e^{-\frac{kn}{m}}\right)^k$$

- Xây dựng bộ lọc nhanh (Fast construction): thực thi một BF rất nhanh, vì nó chỉ đòi hỏi một lần quét của dữ liệu.
- Truy vấn bộ lọc là nhanh và hiệu quả: việc kiểm tra các vị trí của một phần tử trong *S* chỉ yêu cầu tính toán *k* hàm băm (*k* thường một hằng số nhỏ) và truy cập đến *k* bit.

2. Bộ lọc Bloom giao

Bộ lọc Bloom giao (Intersection Bloom Filter, IBF) đã được nhóm đề xuất trong nghiên cứu [17, 41]. IBF là một cấu trúc dữ liệu xác suất được thiết kế để biểu diễn phần giao của hai tập hợp và được sử dụng để nhận ra những phần tử chung của các tập hợp với một xác suất dương tính giả (false positive).

a) Các phương thức xây dựng bộ lọc Bloom giao

IBF tiếp nhận một đầu vào và trả ra một kết quả với một trong hai khả năng có thể xảy ra như sau:

- "*No*": *x* KHÔNG phải là một yếu tố chung của các tập *S1* và *S2*.
- "*Yes*": *x* LÀ một yếu tố chung của các tập *S1* và *S2*.

Ba tiếp cận để xây dựng bộ lọc giao IBF: (1) cặp bộ lọc Bloom, (2) giao của hai bộ lọc Bloom và (3) giao của hai bộ lọc Bloom phân đoạn.

- Tiếp cận 1: Hình 2. Cấu trúc bộ lọc Bloom giao (a) mô tả tiếp cận dùng một cặp bộ lọc Bloom
  Phần giao của hai tập hợp bằng công thức sau:
  $$S1 \cap S2 = (S1 \cup S2) \setminus (S1 \Delta S2) = (S1 \cup S2) \setminus ((S1 \setminus S2) \cup (S2 \setminus S1))$$
  Điều này tương ứng với việc sử dụng bộ lọc BF(*S1*) cho tập *S2* để chọn được các phần tử chung của *S2* và sử dụng bộ lọc BF(*S2*) cho tập *S1* để chọn được các phần tử chung của *S1*. Gộp các kết quả trên chúng ta nhận được tất cả các phần tử chung của cả 2 tập hợp.
  Ưu điểm của cách tiếp cận này là không yêu cầu các bộ lọc phải có cùng kích thước (*m*) và *k* hàm băm.
- Tiếp cận 2: Hình 2. Cấu trúc bộ lọc Bloom giao (b) mô tả tiếp cận giao của hai bộ lọc Bloom

Với cách tiếp cận này đòi hỏi hai bộ lọc Bloom phải có cùng kích thước *m* và *k* hàm băm giống nhau. Để xây dựng được bộ lọc giao nhau IBF, chúng ta tính giao của hai bộ lọc Bloom BF(*S1*) và BF(*S2*) bằng phép toán AND từng bit như sau:

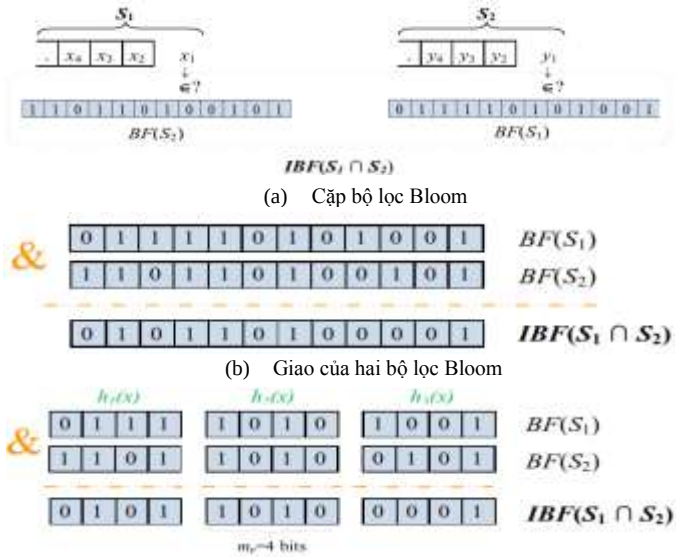
$$IBF(S1, S2) = BF(S1) \,\&\, BF(S2)$$

Để kiểm tra các phần tử có là phần tử chung hay không, tác giả tiến hành thực hiện truy vấn vào bộ lọc giao vừa tạo (IBF). Ưu điểm của phương pháp tiếp cận này là chỉ duy trì một giao nhau lọc Bloom để loại bỏ hầu hết các tuple không tham gia từ hai nguồn dữ liệu đầu vào thay vì sử dụng hai bộ lọc như các phương pháp tiếp cận đầu tiên.

- Tiếp cận 3: Hình 2 (c) mô tả tiếp cận giao của hai bộ lọc Bloom phân đoạn.

Bộ lọc Bloom phân đoạn (Partitioned Bloom filter) [42], là một biến thể của bộ lọc Bloom chuẩn, được định nghĩa bởi một mảng của *m* bit được chia thành *k* mảng con rời nhau với kích thước *mp = m/k* bit.

Tương tự như tiếp cận 2, bộ lọc giao IBF(*S1, S2*) được tạo ra bởi việc giao hai bộ lọc Bloom phân đoạn BF(*S1*) và BF(*S2*) bằng phép toán AND.



Hình 2. Cấu trúc bộ lọc Bloom giao

b) Xác suất phần giao sai

Xác suất phần giao sai chính là tỉ lệ dương tính giả của bộ lọc giao IBF. Từ kết quả đã được chứng minh trong nghiên cứu của chúng tôi [17, 41], xác xuất phần giao sai cho 3 tiếp cận được định nghĩa như sau:

- Xác suất phần giao sai của tiếp cận 1 (dùng cặp bộ lọc Bloom):

a. cho BF(*S1*),

$$f_{\cap pair(S_1)} = \left(1 - \left(1 - \frac{1}{m_1}\right)^{k_1|S_1|}\right)^{k_1}$$

b. cho BF(*S2*),

$$f_{\cap pair(S_2)} = \left(1 - \left(1 - \frac{1}{m_2}\right)^{k_2|S_2|}\right)^{k_2}$$

Trong đó, *m1*, *k1* và *m2*, *k2* lần lượt là là kích thước và số hàm băm tương ứng của bộ lọc BF(*S1*) và BF(*S2*).

- Xác suất phần giao sai của tiếp cận 2 (giao của hai bộ lọc Bloom):

$$f_{\cap BF} = \left(1 - \left(1 - \frac{1}{m}\right)^{k|S_1|}\right)^{k} \left(1 - \left(1 - \frac{1}{m}\right)^{k|S_2|}\right)^{k}$$

Trong đó, $BF(S_1)$, $BF(S_2)$ và $IBF(S_1, S_2)$ có cùng kích thước *m* và *k* hàm băm.

- Xác suất phần giao sai của tiếp cận 3 (giao của hai bộ lọc phân đoạn):

$$f_{\cap PBF} = \left(1 - \left(1 - \frac{k}{m}\right)^{|S_1|}\right)^{k} \left(1 - \left(1 - \frac{k}{m}\right)^{|S_2|}\right)^{k}$$

Trong đó, $BF(S_1)$, $BF(S_2)$ và $IBF(S_1, S_2)$ có cùng kích thước *m* và *k* hàm băm, *k* phân đoạn với cùng kích thước $m_p = m/k$.

- So sánh xác suất phần giao sai của các tiếp cận:

  Xác suất phần giao sai của tiếp cận 1 và 2 là gần bằng nhau và nhỏ hơn xác suất phần giao sai của tiếp cận 3:

$$f_{\cap pair} \approx f_{\cap BF} < f_{\cap PBF}$$

  Điều đó có nghĩa là hiệu suất lọc của tiếp cận 1 và 2 là tốt hơn tiếp cận 3 bởi vì chúng nhận biết sai các phần tử chung ít hơn.

D. Giải thuật Intersection Bloom Join

Intersection Bloom Join [41] là giải thuật Join được cải tiến từ giải thuật Bloom Join với việc sử dụng bộ lọc Bloom giao IBF thay vì dùng bộ lọc Bloom chuẩn. Giải thuật này sẽ dùng bộ lọc Bloom giao để lọc trên cả hai tập dữ liệu nhằm loại bỏ hầu hết những phần tử không tham gia vào Join và giảm đáng kể các chi phí liên quan. Nó đã được chứng minh là hiệu quả hơn so với các thuật giải thuật Join khác [17]. Chính vì vậy mà giải thuật Intersection Bloom Join sẽ là giải thuật chính được sử dụng cho Join đệ quy của nghiên cứu này.

III. TỐI ƯU HÓA JOIN ĐỆ QUY TRONG MAPREDUCE VỚI SPARK

A. Join đệ quy

Join đệ quy của một quan hệ $K(x, y)$ là sự bao đóng bắc cầu của quan hệ K. Nó đòi hỏi một sự khởi tạo và sự lặp lại cho đến khi không có kết quả mới nào được tìm thấy [17]:

$$\begin{cases} \text{(Initialization) } A(x,y) = K(x,y) \\ \text{(Iteration) } A(x,y) = A(x,z) \bowtie K(z,y) \end{cases}$$

B. Tối ưu hóa Join đệ quy trong MapReduce

1. Giải thuật Join đệ quy trong MapReduce

Một trong những giải thuật Join đệ quy phổ biến nhất và phù hợp để triển khai trong môi trường MapReduce gần đây là giải thuật Semi-Naïve [17, 37]. Ngoài ra, nó được áp dụng cho việc tính toán trên các tập dữ liệu quan hệ dạng hàng cột (tabular) mà chúng tôi đang quan tâm và xử lý. Chính vì vậy, nghiên cứu này sẽ tập trung trên việc tối ưu hóa thuật toán Semi-Naïve cho Join đệ quy trong môi trường MapReduce.

Giải thuật Semi-Naïve cho việc tính toán Join đệ quy:

Algorithm 1 – Thuật toán Semi-Naïve cho Join đệ quy

$F_0 = \emptyset$, $\Delta F_0 = K(x,y)$, $i=0$;	(1)
Repeat	(2)
i++;	(3)
$F_{i-1} = (\Delta F_0 \cup ... \cup \Delta F_{i-1})$;	(4)
$\Delta F_i = \prod_{xy}(\Delta F_{i-1} \bowtie_z K) - F_{i-1}$;	(5)
Until $\Delta F_i = \emptyset$;	(6)

Giải thuật này sẽ được thực thi bởi một sự lặp lại của 2 công việc MapReduce: công việc Join (Join job) và công việc xác định tập dữ liệu tăng cường (job for computing incremental dataset).

Tại vòng lặp thứ i, công việc Join sẽ thực thi join của ΔF_{i-1} và K để tạo ra tập O_i. Công việc xác định tập dữ liệu tăng cường sẽ đọc tập dữ liệu O_i và tính $\Delta F_i = F_{i-1} - O_i$ bằng cách loại bỏ các bộ trùng lắp trong O_i với các kết quả trước đó.

2. Đề xuất các giải pháp tối ưu cho Join đệ quy trong MapReduce

Xem xét tại dòng (5) của thuật toán Semi-Naïve, $\Delta F_i = \prod_{xy}(\Delta F_{i-1} \bowtie_z K) - F_{i-1}$, chúng tôi tiến hành đề xuất một số giải pháp tối ưu hóa cho thuật toán khi thực thi trong môi trường MapReduce như sau. (1) *Sử dụng cơ chế xử lý lặp trên bộ nhớ* để làm tăng hiệu quả thực thi cho các công việc kế tiếp nhau. Vấn đề này sẽ được chúng tôi giải quyết bởi việc tận dụng khả năng xử lý lặp của Spark RDD. (2) *Sử dụng cơ chế cache tập dữ liệu K* để giảm thiểu chi phí đọc viết dữ liệu lại nhiều lần bởi vì K là tập dữ liệu không có sự thay đổi trong các lần lặp. Chúng tôi đã tìm thấy giải pháp cho vấn đề này bởi việc sử dụng cơ chế vùng đệm của Spark để thực hiện cache tập K trên bộ nhớ, nếu đầy sẽ tràn xuống đĩa cứng theo thứ tự: MEMORY_AND_DISK_SER. (3*) Loại bỏ dữ liệu dư thừa không tham gia vào Join* để giảm đáng kể chi phí có liên quan đến việc đọc/viết và truyền thông cho dữ liệu vô ích đó. Chúng tôi sử dụng bộ lọc giao IBF để loại bỏ dữ liệu dư thừa này trong công việc Join của ΔF và K. Thuật toán Join được sử dụng trong trường hợp này là thuật toán Intersection Bloom Join [41] như đã trình bày trước đây.

Công việc tiền xử lý

Trước khi thực hiện Join hai tập dữ liệu K với ΔF, hai tập dữ liệu này cần được chuyển thành kiểu dữ liệu của Spark – PairRDD và loại bỏ các cặp khóa/giá trị rỗng. Song song đó, chúng ta xây dựng bộ lọc giao $IBF(K, \Delta F)$ theo như tiếp cận đã đề xuất trước đây.

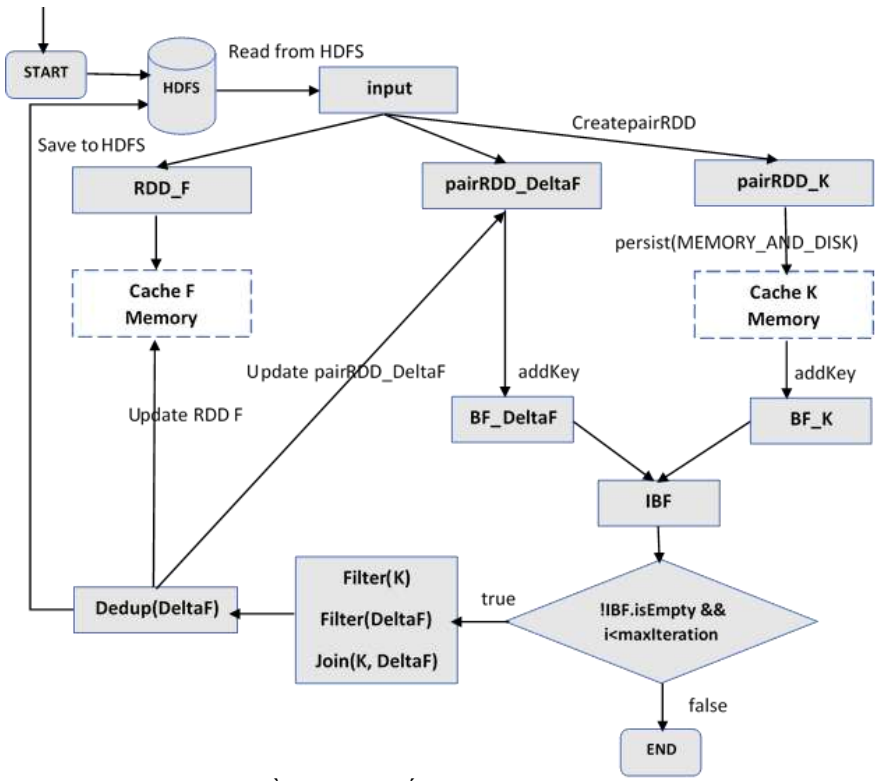
Sự thực thi lặp của Join đệ quy

Hai tập dữ liệu K và ΔF được lọc bởi bộ lọc giao $IBF(K, \Delta F)$ để loại bỏ phần tử không tham gia Join. Sau đó, chúng ta tiến hành Join hai tập dữ liệu đã lọc. Sau mỗi lần join, chúng ta tiến hành xử lý kết quả join để tạo ra ΔF mới bởi công việc thứ 2. Việc join ở lần tiếp theo sẽ thực hiện giữa tập K và ΔF mới vừa tạo ra. Việc join hai tập dữ liệu trên sẽ được lặp lại nhiều lần cho đến khi bắt gặp điều kiện dừng, giải thuật join đệ quy sẽ kết thúc.

Trong quá trình tạo các tập dữ liệu, chúng tôi có thực hiện phân mảnh (PartitionBy) cho các tập dữ liệu, nhằm chia nhỏ dữ liệu, tránh việc tràn bộ nhớ, giảm dữ liệu vận chuyển qua mạng và tăng tốc độ xử lý.

Các điều kiện dừng cho thuật toán Join đệ quy (fixpoint):

- Chu trình xử lý đạt đến số vòng lặp tối đa được giới hạn.
- Hoặc ΔF_i rỗng, không có dữ liệu mới được sinh ra.
- Hoặc giao của hai bộ lọc BF_K và BF_DeltaF_i rỗng, kết quả mới sinh ra không có dữ liệu tham gia vào join ở vòng lặp mới.



Hình 3. Lưu đồ giải thuật tối ưu hoá Join đệ quy trong Spark

Hình 3 là lưu đồ cho giải thuật tối ưu hóa Join đệ quy trong Spark.

Dựa vào giải thuật Semi-Naïve, chúng ta đưa ra giải thuật tối ưu hóa Join đệ quy trong Spark như sau:

Algorithm 2 – Giải thuật tối ưu hóa Semi-Naïve cho Join đệ quy trong MapReduce

// Đọc file từ HDFS
JavaSparkContext.textFile(input)
// Tạo JavaPairRDD
pairRDD_K ← createJavaPairRDD (input, column) *//cache pairRDD_K*
pairRDD_DeltaF ← createJavaPairRDD (input, column)
RDD_F ← createJavaPairRDD (input, column) *//cache RDD_F*
// Xây dựng bộ lọc Bloom

```
BF_K ← addKeyBF_K(pairRDD_K) // BF_K chứa khóa của tập K
BF_DeltaF ← addKeyBF_DeltaF(pairRDD_DeltaF) //BF_DeltaF chứa khóa của DeltaF
// Thực hiện join đệ quy
recursiveJoin (pairRDD_K, pairRDD_DeltaF, iteration){
        IBF ← BF_K.and(BF_DeltaF) //Tạo bộ lọc giao IBF(K, DeltaF)
        if (! IBF.isEmpty() && iteration < maxIteration){
                // Lọc dữ liệu không tham gia join của tập K
                temp_K ← filter(pairRDD_K)
                // Lọc dữ liệu không tham gia join của tập DeltaF
                temp_DeltaF ←filter(pairRDD_DeltaF)
                // Join hai tập dữ liệu đã được lọc
                joinResult ← temp_K.join (temp_DeltaF)
                // Xử lý kết quả join về dạng JavaPairRDD
                rs ← combineKeyValue (JavaPairRDD.fromJavaRDD(joinResult.values(), int
                column);
                //Loại kết quả trùng lắp
                rs ← rs.subtract(RDD_F)
                //Lưu kết quả vào HDFS
                rs.saveAsTextFile (path output)
                pairRDD_DeltaF ← createJavaPairRDD (rs, column)
                // Tạo lại BF_DeltaF
                addKeyBF_DeltaF(pairRDD_DeltaF)
                // Bổ sung kết quả F
                RDD_F ← RDD_F.union(rs) //cache RDD_F
                iteration++
                // Đệ quy
                recursiveJoin (pairRDD_K, pairRDD_DeltaF, iteration)
        }
}
```

3. Mô hình chi phí của Join đệ quy trong MapReduce

a) Các tham số và mô hình chi phí

Bảng 1 chỉ ra bảng các tham số được sử dụng trong mô hình chi phí của Join đệ quy trong Map Reduce.

Bảng 1. Các tham số cho mô hình chi phí của Join đệ quy trong MapReduce

Parameter	Explanation
c_l	Chi phí đọc hay ghi dữ liệu cục bộ
c_r	Chi phí đọc hay ghi dữ liệu từ xa
c_t	Chi phí truyền thông dữ liệu từ một nút đến một nút khác
B+1	Kích thước vùng đệm sắp xếp là B+1 pages
mp_k	Tổng số tác vụ map của tập dữ liệu K
$mp_{\Delta Fi\text{-}1}$	Tổng số tác vụ map của tập dữ liệu tăng cường $\Delta F_{i\text{-}1}$
mp_{Oi}	Tổng số tác vụ map của tập dữ liệu kết quả Join O_i
$\|K\|$	Kích thước của tập dữ liệu K
$\|\Delta F_{i\text{-}1}\|$	Kích thước của tập dữ liệu tăng cường tại lần lặp i-1. $\|\Delta F_0\| = \|K\|$
$\|\Delta F_i\|$	Kích thước của tập dữ liệu tăng cường tại lần lặp i
$\|F_{i\text{-}1}\|$	Kích thước của tất cả các tập dữ liệu tăng cường trong các lần lặp từ 0 đến i-1.
$\|D_i\|$	Kích thước tập dữ liệu trung gian của công việc Join J_i
$\|D^+_i\|$	Kích thước tập dữ liệu trung gian của công việc tính tập dữ liệu tăng cường I_i
$\|O_i\|$	Kích thước tập dữ liệu kết quả của công việc Join tại vòng lặp thứ i
$f_{IBF(K)}$	Xác xuất dương tính giả của bộ lọc giao Bloom $IBF(K)$

$\partial^{i-1}{}_K$	Tỉ lệ Join của ΔF_{i-1} với K
φ_i	Tỉ lệ khác biệt của tập kết quả O_i với F_{i-1}
l	Số lần lặp cũng là chiều sâu nhất của đồ thị quan hệ trừ 1.
C_{pre}	Tổng chi phí thực thi công việc tiền xử lý và phân tán $BF(K)$ đến các nút
C_K	Tổng chi phí để đọc, map, sắp xếp, trộn (shuffle), và cache K tại các the reducers (RIC) trong lần lặp đầu tiên
$C_{read}(J_i)$	Tổng chi phí để đọc tập dữ liệu tăng cường ΔF_{i-1}
$C_{sort}(J_i)$	Tổng chi phí để sắp xếp và sao chép (sort and copy) cho công việc Join tại các nút mappers và reducers
$C_{tr}(J_i)$	Tổng chi phí để chuyển dữ liệu trung gian giữa các nút cho công việc Join
$C_{cache}(J_i)$	Tổng chi phí để đọc cục bộ các phần dữ liệu K được cache tại các reducers
$C_{write}(J_i)$	Tổng chi phí để viết kết quả của công việc Join (O_i)
$C_{read}(I_i)$	Tổng chi phí để đọc O_i
$C_{sort}(I_i)$	Tổng chi phí để sắp xếp và sao chép (sort and copy) cho công việc tính tập dữ liệu tăng cường tại các nút mappers và reducers
$C_{tr}(I_i)$	Tổng chi phí để chuyển dữ liệu trung gian giữa các nút cho công việc tính tập dữ liệu tăng cường
$C_{cache}(I_i)$	Tổng chi phí để đọc cục bộ các phần dữ liệu F_{i-1} được cache tại các reducers
$C_{write}(I_i)$	Tổng chi phí để viết kết quả tập dữ liệu tăng cường ΔF_i

Theo đó, chúng ta có thể đưa ra chi phí tổng của Join đệ quy như sau:

$$C(\hat{J}) = C_K + \sum_{i=1}^{l} C_{read}(J_i) + C_{sort}(J_i) + C_{tr}(J_i) + C_{cache}(J_i) + C_{write}(J_i)$$

$$+ \sum_{i=1}^{l} C_{read}(I_i) + C_{sort}(I_i) + C_{tr}(I_i) + C_{cache}(I_i) + C_{write}(I_i)$$

Trong đó:

- $C_K = c_r . |K| + c_l . |K| . 2 . (\lceil \log_B |K| - \log_B(mp_k) \rceil + \lceil \log_B(mp_k) \rceil) + (c_t + c_l) . |K|$
- $C_{read}(J_i) = c_r . |\Delta F_{i-1}|$
- $C_{sort}(J_i) = c_l . |D_i| . 2 . (\lceil \log_B |D_i| - \log_B(mp_{\Delta Fi-1}) \rceil + \lceil \log_B(mp_{\Delta Fi-1}) \rceil)$
- $C_{tr}(J_i) = c_t . |D_i|$
- $C_{cache}(J_i) = c_l . |K|$
- $C_{write}(J_i) = c_r . |O_i|$
- $|D_i| = |\Delta F_{i-1}| = \varphi_{i-1} . |O_{i-1}|$
- $C_{read}(I_i) = c_r . |O_i|$
- $C_{sort}(I_i) = c_l . |D^+{}_i| . 2 . (\lceil \log_B |O_i| - \log_B(mp_{Oi}) \rceil + \lceil \log_B(mp_{Oi}) \rceil)$
- $C_{tr}(I_i) = c_t . |D^+{}_i|$
- $C_{cache}(I_i) = c_l . |F_{i-1}|$
- $C_{write}(I_i) = c_r . |\Delta F_i|$
- $|D^+{}_i| = |O_i|$
- $|\Delta F_i| = \varphi_i . |O_i|$

b) Phân tích chi phí giữa hai hướng tiếp cận

Đầu tiên, chúng ta so sánh chi phí thực thi Join đệ quy trên nền tảng Hadoop và Spark: $C_{Hadoop}(\hat{J})$ và $C_{Spark}(\hat{J})$. Hầu hết các hoạt động I/O của Hadoop là trên đĩa hoặc từ xa trên HDFS, trong khi chúng được thực thi trên bộ nhớ đối với Spark. Vì vậy, tất cả các chi phí thành phần của Hadoop là lớn hơn nhiều so với chi phí của Spark. Chúng ta dễ dàng có thể suy ra:

$$C_{Hadoop}(\hat{J}) > C_{Spark}(\hat{J})$$

Vấn đề còn lại là chúng ta so sánh chi phí thực thi Join đệ quy chưa tối ưu với giải pháp tối ưu của chúng tôi trên nền tảng Spark: $C_{Spark}(\hat{J})$ và $C_{OptSpark}(\hat{J})$.

Chúng ta cần so sánh thành phần dữ liệu trung gian được tạo ra trong công việc Join bởi vì nó là yếu tố quyết định đến chi phí tổng thực thi Join đệ quy. Lượng dữ liệu trung gian của công việc Join trong Join đệ quy như sau:

- Đối với Join đệ quy chưa tối ưu:

$$|D_i| = |\Delta F_{i-1}| = \varphi_{i-1}.|O_{i-1}|$$

- Đối với Join đệ quy tối ưu:

$$|D_i| = \partial^{i-1}{}_K.|\Delta F_{i-1}| + f_{IBF(K)} \cdot (1 - \partial^{i-1}{}_K).|\Delta F_{i-1}|$$
$$= \partial^{i-1}{}_K.|D_i| + f_{IBF(K)} \cdot (1 - \partial^{i-1}{}_K).\ |D_i| \ \leq\ |D_i|$$

Ngoài ra, giải pháp tối ưu sẽ kết thúc trước 1 vòng lặp so với giải pháp chưa tối ưu nhờ vào đặc tính bộ lọc Bloom giao. Vì vậy, sau cùng chúng ta có thể nhận được biểu thức:

$$C_{Hadoop}(\hat{J}) > C_{Spark}(\hat{J}) > C_{OptSpark}(\hat{J})$$

Điều này cũng có nghĩa là những giải pháp đề xuất trong nghiên cứu này đã mang lại sự tối ưu hóa các chi phí thực thi cho Join đệ quy trong Spark hơn so với các giải pháp hiện tại.

So với giải thuật Join đệ quy chưa cải tiến (không sử dụng cache, bộ lọc), giải thuật Join đệ quy có sử dụng cache và bộ lọc của chúng tôi giảm được lượng chi phí rất lớn trong việc đọc ghi lại dữ liệu nhiều lần trên đĩa, dữ liệu dư thừa trong quá trình Join và dữ liệu vận chuyển qua mạng. Cùng với đó, việc sử dụng bộ lọc giao làm điều kiện dừng cũng tránh được lần Join dư thừa cuối cùng, không cần phải đợi kết quả Join rỗng mới kết thúc chương trình.

IV. THỰC NGHIỆM VÀ ĐÁNH GIÁ

A. Mô tả cluster và dữ liệu

Chúng tôi tiến hành thực nghiệm lần lượt trên 1 cụm máy tính gồm 14 nút (1 master và 13 slaves) tại Phòng thí nghiệm Mạng di động và Dữ liệu lớn của Khoa Công nghệ thông tin và TT, Trường Đại học Cần Thơ. Mỗi máy tính có cấu hình 4 CPUs Intel Core i5 3.2 Ghz với RAM: 4GB, HDD: 500GB, hệ điều hành Ubuntu 14.04 LTS 64 bits. Phiên bản Java được cài đặt là 1.8, Hadoop 2.7.1, Spark 1.6.

Dữ liệu chuẩn được sinh ra bởi Purdue MapReduce Benchmarks Suite có kích thước 5GB, 10GB, 20GB và 30GB. Các tập dữ liệu đều được lưu trữ dạng tập tin văn bản (text file). Mỗi dòng trong tập tin dữ liệu có tối đa 39 trường phân biệt bởi dấu phẩy. Mỗi trường dữ liệu chứa 19 ký tự.

B. Phương thức đánh giá

Nhóm chúng tôi áp dụng ba tiếp cận: giải thuật Semi-Naïve thuần trên Hadoop, giải thuật Semi-Naïve thuần trên Spark và Semi-Naïve có áp dụng những giải pháp cải tiến của chúng tôi để thực thi câu truy vấn Join đệ quy như đã đề cập trong phần I. Trên mỗi tập dữ liệu thực nghiệm, chúng tôi đánh giá ba tiếp cận dựa trên lượng dữ liệu trung gian cần vận chuyển qua mạng, lượng dữ liệu cần đọc vào và thời gian thực thi.

C. Tham số sử dụng cho bộ lọc Bloom

Các tham số dùng để xây dựng bộ lọc Bloom được chỉ ra như trong bảng sau:

Bảng 2. Các Tham số để xây dựng bộ lọc Bloom

Tests	f_{BF0}	n_0	k	m/n	m (bit)	n	f_{BF}
Test 1	0.000101	50,000	8	21	1,050,000	15,002	0.000000
Test 2	0.000101	50,000	8	21	1,050,000	15,053	0.000000
Test 3	0.000101	50,000	8	21	1,050,000	15,101	0.000000
Test 4	0.000101	50,000	8	21	1,050,000	15,121	0.000000

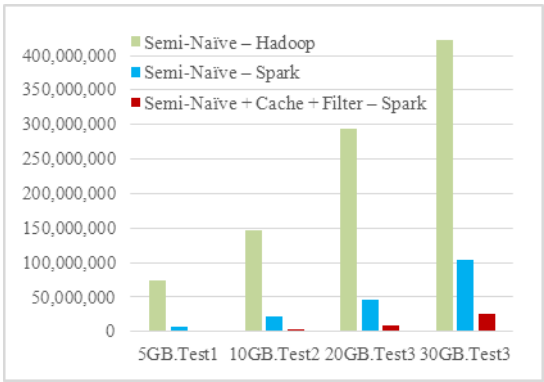
D. Đánh giá ba hướng tiếp cận

Chúng tôi tiến hành so sánh giải thuật Semi-Naïve trên Hadoop, Semi-Naïve trên Spark và Semi-Naïve đã cải tiến để thấy được lượng dữ liệu trung gian, lượng dữ liệu đọc vào và thời gian thực thi, qua đó đánh giá mức độ cải tiến của đề xuất trong nghiên cứu này.

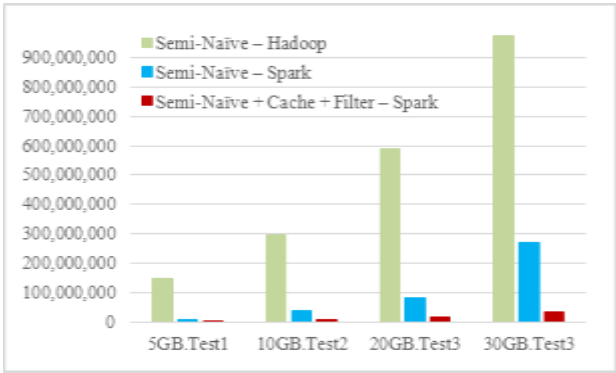
Bảng 3. Lượng dữ liệu trung gian của các tiếp cận (#records)

Tiếp cận	5GB.Test1	10GB.Test2	20GB.Test3	30GB.Test3
Semi-Naïve – Hadoop	73,339,332	146,668,049	293,330,613	421,466,111
Semi-Naïve – Spark	7,281,636	22,884,615	45,755,154	103,957,219
Semi-Naïve + Cache + Filter – Spark	1,139,985	4,158,931	9,152,159	24,870,278

Bảng 3 mô tả kết quả lượng dữ liệu trung gian cần vận chuyển qua mạng giữa ba hướng tiếp cận:



Hình 5. Lượng dữ liệu trung gian giữa ba hướng tiếp cận



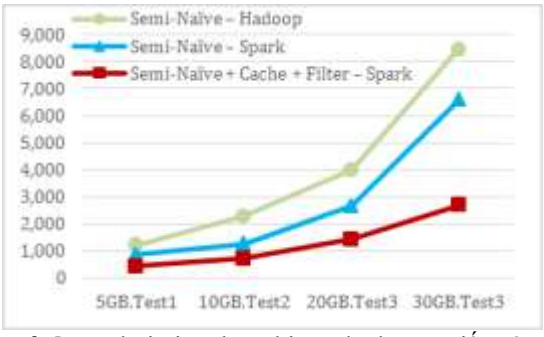
Hình 5. Lượng dữ liệu cần đọc của ba hướng tiếp cận

Kết quả so sánh thể hiện rất rõ về các cải tiến của Spark so với Hadoop trong Hình 4. Từ việc quản lý công việc lặp và phân chia các partitions trên bộ nhớ giúp giảm thiểu đáng kể lượng dữ liệu trung gian cần vận chuyển qua mạng. Bộ lọc Bloom và tính năng cache đã tiếp tục cải tiến, làm giảm thêm lượng dữ liệu dư thừa không tham gia vào quá trình Join, tối ưu được việc xử lý cho join đệ quy trên Spark.

Bảng 4 mô tả kết quả lượng dữ liệu cần đọc giữa ba hướng tiếp cận.

Bảng 4. Lượng dữ liệu cần đọc của các tiếp cận

Tiếp cận (s)	5GB.Test1	10GB.Test2	20GB.Test3	30GB.Test3
Semi-Naïve – Hadoop	147,615,684	295,213,667	590,437,461	975,000,351
Semi-Naïve – Spark	9,362,202	41,615,177	83,208,544	268,632,499
Semi-Naïve + Cache + Filter – Spark	2,080,330	8,317,760	16,627,752	34,850,020



Hình 6. Thời gian thực thi của ba hướng tiếp cận

Kết quả từ Hình 5 cho thấy từ tính năng lập lịch biểu để xử lý luồng dữ liệu và quản lý vòng lặp, Spark đã thực thi trực tiếp trên bộ nhớ tránh được việc quét lại dữ liệu nhiều lần. Bên cạnh đó, việc sử dụng cache đã tối ưu hơn nữa chi phí quét dữ liệu đầu vào.

Hình 6 cho thấy Spark cải thiện được rất nhiều về tốc độ xử lý so với Hadoop. Từ các nghiên cứu cải tiến đã giúp tối ưu hoá hơn nữa thời gian thực thi cho giải thuật Semi Naïve. Đối với giải thuật Semi-Naïve có cải tiến, dữ liệu càng lớn thì hiệu suất càng được cải thiện, dữ liệu nhỏ sẽ tốn chi phí cho việc xử lý bộ lọc.

Bảng 5. Thời gian thực thi của các hướng tiếp cận (giây)

Tiếp cận (s)	5GB.Test1	10GB.Test2	20GB.Test3	30GB.Test3
Semi-Naïve – Hadoop	1,202	2,279	3,984	8,472
Semi-Naïve – Spark	844	1,251	2,669	6,583
Semi-Naïve + Cache + Filter – Spark	424	715	1,436	2,696

V. KẾT LUẬN VÀ KIẾN NGHỊ

A. Kết luận nghiên cứu

Nhóm nghiên cứu đã đưa ra những giải pháp cải tiến hiệu quả cho vấn đề "Tối ưu hoá Join đệ quy trên tập dữ liệu lớn trong môi trường MapReduce". Những kết quả đáng chú ý của nghiên cứu này bao gồm:

(1) Một điều tra các giải pháp hiện có cho Join đệ quy trên tập dữ liệu lớn trong môi trường MapReduce. Nó cũng chỉ ra những hạn chế của các giải pháp và sự cần thiết cho những đề xuất của nghiên cứu này.

(2) Tối ưu hóa cho Join đệ quy trong MapReduce. Những giải pháp hiệu quả được dùng để cải tiến Join đệ quy như sau: (a) Cơ chế xử lý lặp trên bộ nhớ với Spark RDD nhằm làm tăng hiệu quả thực thi; (b) Cơ chế vùng nhớ đệm của Spark nhằm cache tập dữ liệu không đổi trong các lần lặp và giảm thiểu chi phí đọc viết lại dữ liệu; (c) Bộ lọc giao và thuật toán Intersection Bloom Join nhằm loại bỏ dữ liệu dư thừa không tham gia vào Join, cũng có nghĩa là làm giảm chi phí có liên quan đến việc đọc/viết và truyền thông cho dữ liệu vô ích.

(3) Mô hình chi phí cho Join đệ quy trong MapReduce. Đây là cơ sở lý thuyết quan trọng để làm cơ sở cho việc đánh giá và so sánh các giải pháp.

(4) Sự triển khai và phát triển ứng dụng trên các nền tảng xử lý dữ liệu lớn phổ biến nhất hiện nay như Hadoop và thế hệ mới mới Spark.

(5) Các thực nghiệm và đánh giá cho Join đệ quy trong MapReduce với Hadoop và Spark.

Join đệ quy là một phép toán tiêu tốn nhiều chi phí, thời gian và tài nguyên. Thông qua mô hình chi phí và các thực nghiệm, chúng tôi đã chứng minh được rằng các giải pháp cải tiến của chúng tôi đã mang lại hiệu quả đáng kể hơn so với giải pháp hiện nay trong tính toán Join đệ quy trên tập dữ liệu lớn. Việc thực thi Join đệ quy trong môi trường Spark của chúng tôi đã khai thác được tối đa khả năng của Spark như xử lý song song phân tán, xử lý lặp, cơ chế bộ nhớ đệm và tính toán nhanh trên bộ nhớ. Hơn nữa, việc sử dụng bộ lọc Bloom để loại bỏ các phần tử không tham gia join của toàn bộ tập dữ liệu đầu vào đã làm giảm bớt gánh nặng đọc/viết và xử lý quá nhiều dữ liệu.

Những đóng góp của chúng tôi có ý nghĩa thực tiễn cao. Bộ lọc giao có thể được áp dụng để giải quyết các vấn đề phổ biến trong nhiều lĩnh vực như Join, sự hòa giải và chống trùng lắp dữ liệu (reconciliation and deduplication), sửa lỗi (error-correction), v.v. Tối ưu hóa Join đệ quy trong MapReduce với Spark mang lại lợi ích to lớn cho nhiều lĩnh vực như cơ sở dữ liệu lớn, mạng xã hội, tin sinh học, mạng sensor, giám sát mạng, máy học, v,v. Sau cùng, nghiên cứu này là bước đi quan trọng đóng góp vào ngữ cảnh tối ưu hóa quản lý dữ liệu lớn trên cơ sở hạ tầng đám mây.

B. Hướng phát triển

Mặc dù, giải thuật đã giảm thiểu nhiều chi phí cho việc đọc/viết dữ liệu và tăng tốc cho quá trình xử lý. Tuy nhiên, việc tối ưu Join đệ quy trên tập dữ liệu lớn trong môi trường Spark vẫn còn tồn tại nhiều hạn chế. Để đạt được hiệu quả như mong muốn, đòi hỏi phải có một hệ thống cụm máy tính (cluster) đủ mạnh và chạy ổn định xử lý dữ liệu sau mỗi lần Join. Hơn nữa, vấn đề nghiêng dữ liệu là một thách thức rất lớn cho đề tài này và cho bài toán xử lý dữ liệu lớn nói chung. Đây cũng là hướng phát triển mà đề tài muốn hướng đến trong tương lai.

TÀI LIỆU THAM KHẢO

[1] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, p. 107, Jan. 2008.

[2] "Apache Hive TM." [Online]. Available: https://hive.apache.org/. [Accessed: 14-Jun-2016].

[3] K. Wiley, A. Connolly, S. Krughoff, J. Gardner, M. Balazinska, B. Howe, Y. Kwon, and Y. Bu, "Astronomical Image Processing with Hadoop," *ResearchGate*, Jul. 2011.

[4] Page, Lawrence, Brin, Sergey, Motwani, Rajeev, Winograd, and Terry, "Page, L., et al.: The PageRank citation ranking: Bringing order to the web," *ResearchGate*, Jan. 1998.

[5] J. M. Kleinberg, "Authoritative Sources in a Hyperlinked Environment," *J ACM*, vol. 46, no. 5, pp. 604–632, Sep. 1999.

[6] Bancilhon and R. Ramakrishnan, "An Amateur's Introduction to Recursive Query Processing Strategies," 1986. .

[7] A. K. Jain, M. N. Murty, and P. J. Flynn, *Data Clustering: A Review*. 1999.

[8] M. T. Hagan, H. B. Demuth, M. H. Beale, and O. De Jess, *Neural Network Design*, 2nd ed. USA: Martin Hagan, 2014.

[9] S. Wasserman and K. Faust, *Social Network Analysis: Methods and Applications*. Cambridge University Press, 1994.

[10] A. W. Moore and D. Zuev, "Internet Traffic Classification Using Bayesian Analysis Techniques," in *Proceedings of the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, New York, NY, USA, 2005, pp. 50–60.

[11] E. F. Codd, "A Relational Model of Data for Large Shared Data Banks," *Commun ACM*, vol. 13, no. 6, pp. 377–387, Jun. 1970.

[12] E. F. Codd, "Relational Completeness of Data Base Sublanguages," *R Rustin Ed Database Syst. 65-98 Prentice Hall IBM Res. Rep. RJ 987 San Jose Calif.*, 1972.

[13] K.-L. Tan and H. Lu, "A Note on the Strategy Space of Multiway Join Query Optimization Problem in Parallel Systems," *SIGMOD Rec*, vol. 20, no. 4, pp. 81–82, Dec. 1991.

[14] X. Lin and M. E. Orlowska, "An efficient processing of a chain join with the minimum communication cost in distributed database systems," *Distrib. Parallel Databases*, vol. 3, no. 1, pp. 69–83, Jan. 1995.

[15] C. Ordonez, "Optimizing Recursive Queries in SQL," in *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, New York, NY, USA, 2005, pp. 834–839.

[16] S. Idreos, E. Liarou, and M. Koubarakis, "Continuous Multi-way Joins over Distributed Hash Tables," in *Proceedings of the 11th International Conference on Extending Database Technology: Advances in Database Technology*, New York, NY, USA, 2008, pp. 594–605.

[17] T.-C. Phan, L. d'Orazio, and P. Rigaux, "A Theoretical and Experimental Comparison of Filter-Based Equijoins in MapReduce," in *Transactions on Large-Scale Data- and Knowledge-Centered Systems XXV*, A. Hameurlain, J. Küng, and R. Wagner, Eds. Springer Berlin Heidelberg, 2016, pp. 33–70.

[18] "Apache Spark™ - Lightning-Fast Cluster Computing." [Online]. Available: http://spark.apache.org/. [Accessed: 14-Jun-2016].

[19] "The Apache Cassandra Project." [Online]. Available: http://cassandra.apache.org/. [Accessed: 14-Jun-2016].

[20] "Apache HBase – Apache HBase™ Home." [Online]. Available: https://hbase.apache.org/. [Accessed: 14-Jun-2016].

[21] "Amazon Simple Storage Service (S3) - Cloud Storage," *Amazon Web Services, Inc.* [Online]. Available: //aws.amazon.com/s3/. [Accessed: 14-Jun-2016].

[22] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster Computing with Working Sets," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, Berkeley, CA, USA, 2010, pp. 10–10.

[23] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, Berkeley, CA, USA, 2012, pp. 2–2.

[24] F. Bancilhon, "Naive Evaluation of Recursively Defined Relations," in *On Knowledge Base Management Systems*, M. L. Brodie and J. Mylopoulos, Eds. Springer New York, 1986, pp. 165–178.

[25] I. Balbin and K. Ramamohanarao, "A Generalization of the Differential Approach to Recursive Query Evaluation," *J Log Program*, vol. 4, no. 3, pp. 259–262, Sep. 1987.

[26] F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman, "Magic Sets and Other Strange Ways to Implement Logic Programs (Extended Abstract)," in *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, New York, NY, USA, 1986, pp. 1–15.

[27] J. D. Ullman, *Principles of Database and Knowledge-base Systems, Vol. I*. New York, NY, USA: Computer Science Press, Inc., 1988.

[28] Y. E. Ioannidis, "On the Computation of the Transitive Closure of Relational Operators," in *Proceedings of the 12th International Conference on Very Large Data Bases*, San Francisco, CA, USA, 1986, pp. 403–411.

[29] P. Valduriez and H. Boral, "Evaluation of Recursive Queries Using Join Indices," in *Expert Database Systems*, 1986, pp. 271–293.

[30] S. Warshall, "A Theorem on Boolean Matrices," *J ACM*, vol. 9, no. 1, pp. 11–12, Jan. 1962.

[31] H. S. Warren Jr., "A Modification of Warshall's Algorithm for the Transitive Closure of Binary Relations," *Commun ACM*, vol. 18, no. 4, pp. 218–220, Apr. 1975.

[32] F. N. Afrati, V. Borkar, M. Carey, N. Polyzotis, and J. D. Ullman, "Map-reduce Extensions and Recursive Queries," in *Proceedings of the 14th International Conference on Extending Database Technology*, New York, NY, USA, 2011, pp. 1–8.

[33] F. N. Afrati, V. Borkar, M. Carey, N. Polyzotis, and J. D. Ullman, "Cluster Computing, Recursion and Datalog," in *Proceedings of the First International Conference on Datalog Reloaded*, Berlin, Heidelberg, 2011, pp. 120–144.

[34] G. Malewicz, M. H. Austern, A. J. . Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A System for Large-scale Graph Processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, New York, NY, USA, 2010, pp. 135–146.

[35] J. Chandar, "Join Algorithms using Map/Reduce," University of Edinburgh, 2010.

[36] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "The HaLoop Approach to Large-scale Iterative Data Analysis," *VLDB J.*, vol. 21, no. 2, pp. 169–190, Apr. 2012.

[37] T. T. Q. Tran, "Traitement de la jointure récursive en MapReduce," Université Blaise Pascal-Clermont-Ferrand II, Clermont-Ferrand, 2014.

[38] M. Shaw, P. Koutris, B. Howe, and D. Suciu, "Optimizing Large-scale Semi-NaïVe Datalog Evaluation in Hadoop," in *Proceedings of the Second International Conference on Datalog in Academia and Industry*, Berlin, Heidelberg, 2012, pp. 165–176.

[39] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970.

[40] D. Guo, J. Wu, H. Chen, and X. Luo, "Theory and Network Applications of Dynamic Bloom Filters," in *Proceedings IEEE INFOCOM 2006. 25TH IEEE International Conference on Computer Communications*, 2006, pp. 1–12.

[41] T. C. Phan, L. d'Orazio, and P. Rigaux, "Toward Intersection Filter-based Optimization for Joins in MapReduce," in *Proceedings of the 2Nd International Workshop on Cloud Intelligence*, New York, NY, USA, 2013, p. 2:1–2:2.

[42] A. Kirsch and M. Mitzenmacher, "Less Hashing, Same Performance: Building a Better Bloom Filter," in *Algorithms – ESA 2006*, Y. Azar and T. Erlebach, Eds. Springer Berlin Heidelberg, 2006, pp. 456–467.

OPTIMIZATION FOR RECURSIVE JOINS ON LARGE-SCALE DATASETS USING SPARK

Phan Thuong Cang, Tran Thi To Quyen, Phan Anh Cang

ABSTRACT*— MapReduce has recently become the dominant programming model for analyzing and processing large-scale data. However, the model has its own limitations. It does not completely support iterative computation, caching mechanism, and operations with multiple inputs. Besides, I/O and communication costs of the model are so expensive. One of notably complex operations used extensively and expensively in MapReduce is a recursive Join. It requires processing characteristics that are also the limitations of the MapReduce environment. Therefore, this research proposed efficient solutions of processing the recursive join on large-scale datasets using Spark, a next-generation processing engine of MapReduce. Our proposal eliminates a large amount of redundant data generated in repeated processing of the join, and takes advantages of in-memory computing means and cache mechanism. Through cost models and experiments, the present research shows that our solutions significantly improve the execution perfomance of the recursive Join in MapReduce.*