Tạp chí Khoa học Đại học Cần Thơ
Phần A: Khoa học tự nhiên, Công nghệ và Môi trường
ISSN 1859-2333 | e-ISSN 2815-5599

*DOI:10.22144/ctujos.2024.426*

# ĐÁNH GIÁ CÁC THUẬT TOÁN LỌC HIỆU QUẢ TRONG XỬ LÝ DỮ LIỆU LỚN

Phan Thượng Cang*, Trần Thị Tố Quyên và Triệu Thanh Ngoan
*Trường Công nghệ Thông tin và Truyền thông, Trường Đại học Cần Thơ, Việt Nam*
**Tác giả liên hệ (Corresponding author): ptcang@ctu.edu.vn*

***Thông tin chung (Article Information)***

*Nhận bài (Received): 05/06/2024*
*Sửa bài (Revised): 24/06/2024*
*Duyệt đăng (Accepted): 22/09/2024*

***Title:*** *Evaluating effective filtering algorithms in big data processing*

***Author(s):*** *Phan Thuong Cang*, Tran Thi To Quyen and Trieu Thanh Ngoan*

***Affiliation(s):*** *College of Information and Communication Technology, Can Tho University, Viet Nam*

## TÓM TẮT

*Việc xử lý và phân tích dữ liệu nhanh chóng, hiệu quả trong kỷ nguyên dữ liệu lớn là thách thức quan trọng. Các thuật toán lọc giúp tăng hiệu suất xử lý dữ liệu lớn bằng cách loại bỏ dữ liệu không liên quan, giảm chi phí tính toán, rút ngắn thời gian xử lý truy vấn. Nghiên cứu này đánh giá hiệu năng của 5 thuật toán lọc phổ biến bao gồm Bloom Filter, Cuckoo Filter, Quotient Filter, Morton Filter và Vacuum Filter trong môi trường Apache Spark. Thông qua thực nghiệm trên các tập dữ liệu lớn, kết quả cho thấy Quotient Filter hiệu quả nhất về lưu trữ, Cuckoo Filter thể hiện sự cân bằng tốt giữa tốc độ chèn, tìm kiếm và xóa. Bloom Filter phù hợp với dữ liệu tĩnh, Morton Filter nổi trội về tốc độ tìm kiếm, Vacuum Filter có thời gian chèn chậm nhưng tìm kiếm và xóa nhanh. Việc kết hợp các thuật toán này với Apache Spark giúp cải tiến đáng kể hiệu suất xử lý nhờ khả năng phân tán và song song. Kết quả nghiên cứu cung cấp lựa chọn thuật toán lọc phù hợp và chỉ ra tiềm năng ứng dụng hiệu quả các thuật toán lọc trong xử lý dữ liệu quy mô lớn.*

***Từ khóa:*** *Apache Spark, Bloom Filter, Cuckoo Filter, Morton Filter, Quotient Filter, Vacuum Filter*

## ABSTRACT

*Handling and analyzing data quickly and efficiently in the era of big data is a significant challenge. Filtering algorithms enhance the performance of big data processing by eliminating irrelevant data, reducing computational costs, and shortening query processing times. This study evaluates the performance of five popular filtering algorithms: Bloom Filter, Cuckoo Filter, Quotient Filter, Morton Filter, and Vacuum Filter in an Apache Spark environment. Through experiments on large datasets, the results show that the Quotient Filter is the most efficient in terms of storage, the Cuckoo Filter demonstrates a good balance between insertion, search, and deletion speeds. The Bloom Filter is suitable for static data, the Morton Filter excels in search speed, and the Vacuum Filter has a slow insertion time but fast search and deletion times. Integrating these algorithms with Apache Spark significantly improves processing performance thanks to its distributed and parallel capabilities. The study results provide guidance on selecting suitable filtering algorithms and highlight the potential for effectively applying filtering algorithms in large-scale data processing.*

***Keywords:*** *Apache Spark, Bloom Filter, Cuckoo Filter, Morton Filter, Quotient Filter, Vacuum Filter*

## 1. GIỚI THIỆU

Trong kỷ nguyên dữ liệu lớn, việc xử lý và phân tích dữ liệu nhanh chóng và hiệu quả là thách thức lớn (Kumar et al., 2023). Khả năng lọc và truy xuất thông tin liên quan từ tập dữ liệu khổng lồ trở nên quan trọng trong nhiều ứng dụng như khai phá dữ liệu, học máy, phân tích thời gian thực và hỗ trợ ra quyết định. Vì vậy, nghiên cứu và tối ưu hóa các thuật toán lọc đã thu hút sự quan tâm lớn của cộng đồng khoa học và chuyên gia ngành. Các thuật toán lọc giúp cải thiện đáng kể hiệu suất xử lý dữ liệu lớn bằng cách loại bỏ dữ liệu không liên quan hoặc dư thừa, giảm chi phí tính toán và rút ngắn thời gian truy vấn (Li, 2021). Hiệu quả của chúng tác động trực tiếp tới hiệu năng tổng thể và khả năng mở rộng của hệ thống xử lý dữ liệu lớn, khiến việc tối ưu hóa thuật toán lọc trở thành lĩnh vực nghiên cứu quan trọng (García et al., 2016). Các thuật toán lọc phổ biến như Bloom Filter, Cuckoo Filter, Quotient Filter, Morton Filter và Vacuum Filter đều có ưu nhược điểm riêng về hiệu quả lưu trữ, tốc độ chèn và truy vấn, tỷ lệ dương tính giả và khả năng thích ứng với đặc tính dữ liệu khác nhau. Việc hiểu rõ sự đánh đổi của từng thuật toán và lựa chọn giải pháp phù hợp cho từng bài toán cụ thể là rất quan trọng để tối ưu hóa hiệu suất xử lý.

Bên cạnh đó, sự phát triển của các framework tính toán phân tán như Apache Spark đã tạo ra cuộc cách mạng trong xử lý dữ liệu lớn thông qua việc cho phép thực thi song song và phân tán các thuật toán lọc trên cụm máy tính lớn. Tích hợp các thuật toán lọc với các framework này hứa hẹn tăng tốc đáng kể tốc độ xử lý và khả năng xử lý khối lượng dữ liệu ngày càng tăng. Tuy nhiên, việc đánh giá và phân tích hiệu năng của các thuật toán lọc trong môi trường phân tán là cần thiết do sự khác biệt với môi trường truyền thống. Nhiều nghiên cứu gần đây đã tập trung vào ứng dụng và cải tiến các bộ lọc cấu trúc dữ liệu xác suất trong nhiều lĩnh vực khác nhau. Maulana et al. (2023) đề xuất sử dụng bộ lọc Bloom, Xor và Cuckoo để tối ưu hóa truy vấn cơ sở dữ liệu cho doanh nghiệp vừa và nhỏ. Ezzaki et al. (2020) cung cấp tổng quan về các biến thể của bộ lọc Bloom. Burdakov et al. (2019) áp dụng Bloom Filter Cascade trên Spark để tối ưu truy vấn SQL. Một số nghiên cứu khác tập trung vào bài toán tìm kiếm và kết hợp tương đồng chuỗi sử dụng các phương pháp lọc (Chaudhuri et al., 2006; Yu et al., 2016; Yan et al., 2017; Fier et al., 2018; Tran et al., 2020; Li, 2021). Các nghiên cứu này chỉ ra sự phát triển tích cực và tiềm năng ứng dụng rộng rãi của các phương pháp lọc dữ liệu.

Nghiên cứu này nhằm giải quyết nhu cầu cấp thiết về đánh giá và so sánh toàn diện hiệu năng của các thuật toán lọc khác nhau trong bối cảnh xử lý dữ liệu lớn. Thông qua thử nghiệm và so sánh mở rộng, việc đánh giá ưu nhược điểm của từng thuật toán và sự phù hợp của chúng trong các tình huống cụ thể được thực hiện; đồng thời, nghiên cứu tác động của việc tích hợp các thuật toán này với Apache Spark, chỉ ra tiềm năng cải thiện hiệu suất đáng kể thông qua xử lý song song và phân tán. Những đóng góp chính của bài báo bao gồm: (1) tổng quan và phân tích toàn diện về 5 thuật toán lọc gồm Bloom Filter, Cuckoo Filter, Quotient Filter, Morton Filter và Vacuum Filter; (2) đánh giá hiệu năng thực nghiệm của các thuật toán trên tập dữ liệu thực tế; (3) phân tích ảnh hưởng của việc tích hợp các thuật toán với Apache Spark tới khả năng cải thiện hiệu suất; (4) đưa ra hướng dẫn lựa chọn thuật toán lọc phù hợp dựa trên yêu cầu và đặc điểm dữ liệu của ứng dụng. Các kết quả của nghiên cứu hứa hẹn giúp các chuyên gia tối ưu hóa quy trình xử lý dữ liệu lớn.

## 2. CƠ SỞ LÝ THUYẾT

### 2.1. Bloom Filter

Bộ lọc Bloom (Lu et al., 2005) được giới thiệu bởi Burton Howard Bloom vào năm 1970, là một cấu trúc dữ liệu xác suất (PDS) được sử dụng rộng rãi trong lĩnh vực truy xuất và lưu trữ thông tin. Bộ lọc này dùng để kiểm tra tư cách thành viên của một phần tử trong tập hợp, xác nhận chắc chắn về việc không phải là thành viên và tư cách thành viên được suy ra theo xác suất với một sai số có thể tính toán được, gọi là tỷ lệ dương tính giả. Bộ lọc Bloom sử dụng một mảng B gồm m bit và một tập hợp S = x1,x2,x3,...,xn gồm n phần tử. Ngoài ra, k hàm băm h1,h2,...,hk được dùng để ánh xạ mỗi phần tử trong S thành m vị trí trong mảng B, với k được tính bằng công thức k = (m/n)log(2). Ban đầu, tất cả các bit trong mảng B được đặt bằng 0.

**Thuật toán 1:** Bloom Filter

```
procedure Initialize(m, k)
   B ← array of m bits, all set to 0
   H ← array of k hash functions
end procedure
procedure Add(x)
   for i ← 1 to k do
      position ← Hi(x) mod m
      B[position] ← 1
   end for
end procedure
procedure Query(x)
   for i ← 1 to k do
```

```
    position ← Hi(x) mod m
    if B[position] = 0 then
      return False
    end if
  end for
  return True
end procedure
```

### 2.2. Cuckoo Filter

Bộ lọc Cuckoo (Fan et al., 2014) là một cấu trúc dữ liệu mạnh mẽ cho các truy vấn về tư cách thành viên tập hợp gần đúng, mang lại nhiều ưu điểm so với bộ lọc Bloom truyền thống. Khác với bộ lọc Bloom chỉ hỗ trợ thêm và truy vấn, bộ lọc Cuckoo cho phép thêm, xóa và truy vấn phần tử một cách linh hoạt. Giả sử cho tập hợp S = x1,x2,x3,...,xn gồm n phần tử, bộ lọc Cuckoo sử dụng mảng CF gồm m bucket, mỗi bucket chứa nhiều slot (thường là 4). Hai hàm băm h1 và h2 được dùng để ánh xạ các phần tử từ S đến các vị trí trong CF. Ngoài ra, hàm băm f(x) được sử dụng để tạo mã băm (fingerprint) cho phần tử x trong S.

**Thuật toán 2:** Cuckoo Filter

```
procedure Initialize(m, f, d)
  T ← array of m buckets, each with f slots, all set to empty
  k ← number of hash functions
  h ← array of k hash functions
end procedure
procedure Add(x)
  i ← 0
  while i < k do
    index ← hi(x)
    if T[index] has an empty slot then
      Insert x into T[index]
      return
    end if
    i ← i + 1
  end while
  Perform cuckoo evictions or resize the filter
end procedure
procedure Search(x)
  i ← 0
  while i < k do
    index ← hi(x)
    if x is in T[index] then
      return True
    end if
    i ← i + 1
  end while
  return False
end procedure
procedure Delete(x)
  i ← 0
  while i < k do
    index ← hi(x)
    if x is in T[index] then
      Remove x from T[index]
      return
    end if
    i ← i + 1
  end while
  return False
end procedure
```

### 2.3. Quotient Filter

Quotient Filter (QF) (Geil et al., 2018) là một cấu trúc dữ liệu xác suất hiệu quả để kiểm tra sự hiện diện của các phần tử trong tập dữ liệu lớn. QF là biến thể của bộ lọc Bloom, chia khóa thành thương số và số dư để sử dụng bộ nhớ hiệu quả hơn (Al-hisnawi & Ahmadi, 2016). QF đạt tỷ lệ nén cao, giảm không gian lưu trữ và lỗi dương tính giả so với bộ lọc Bloom. Tuy nhiên, QF có độ phức tạp cao hơn khi chèn, xóa do cần sắp xếp lại phần tử. QF sử dụng hàm f(x) để tạo mã băm độ dài p bit cho phần tử x. Mã băm gồm số dư fr(x) (r bit trọng số thấp) và thương số fq(x) (q bit trọng số cao), với q = p − r. QF hoạt động như bảng băm mở, gồm 2q bucket, mỗi bucket chứa số dư fr(x) được xác định bởi thương số fq(x). Mỗi bucket có r bit cho số dư và 3 bit siêu dữ liệu: is_occupied, is_continuation, is_shifted. Khi nhiều phần tử có cùng thương số, chúng được sắp xếp thành chuỗi bucket gọi là run. Một cluster là chuỗi các run liên tiếp không có bucket trống ở giữa. Số dư có thể được lưu trong canonical bucket hoặc một bucket gần đó. Khi thêm phần tử x, ta xác định mã băm f(x). Giả sử f(x) = 00000010 (8 bit), cần ít nhất 3 bit cho thương số, 5 bit cho số dư. Khi đó, xq = 000 (thương số) và xr = 00010 (số dư). Vị trí chèn x là 0 vì xq = 000. Giá trị xr = 00010 được thêm vào vị trí 0 trong mảng QF, bit is_occupied được đặt là 1.

**Thuật toán 3:** Quotient Filter

```
procedure Initialize(m, f, q)
  T ← array of m buckets, each with f slots, all set to empty
  Q ← array of q quotient values, initially all set to 0
  k ← number of hash functions
  h ← array of k hash functions
end procedure

procedure Add(x)
  i ← 0
  while i < k do
```

```
        index ← hi(x)
        if T[index] has an empty slot then
            Insert x into T[index]
            return
        end if
        i ← i + 1
    end while
    Perform quotient filter splitting or resizing
end procedure
procedure Search(x)
    i ← 0
    while i < k do
        index ← hi(x)
        if x is in T[index] then
            return True
        end if
        i ← i + 1
    end while
    return False
end procedure

procedure Delete(x)
    i ← 0
    while i < k do
        index ← hi(x)
        if x is in T[index] then
            Remove x from T[index]
            return
        end if
        i ← i + 1
    end while
    return False
end procedure
```

### 2.4. Morton Filter

Bộ lọc Morton Filter (MF) (Breslow & Jayasena, 2018) là một biến thể của bộ lọc Cuckoo với một số cải tiến đáng kể. MF sử dụng kỹ thuật nén dữ liệu, tách biệt đại diện dữ liệu logic và vật lý, cũng như ưu tiên một số bucket để tối ưu hóa hiệu suất và sử dụng bộ nhớ. MF nén các bucket và chỉ lưu trữ các mã băm khác rỗng, giúp giảm đáng kể số lần truy cập bộ nhớ cho mỗi hoạt động. MF sử dụng tập hợp MF với n bucket, mỗi bucket chứa nhiều slot. Hàm f(x) là mã băm của phần tử x, h1(x) và h2(x) ánh xạ x vào một vị trí bucket trong MF. f(x) được lưu tại vị trí do h1(x) hoặc h2(x) xác định.

**Thuật toán 4:** Morton Filter

```
procedure Initialize(m, n, k)
    MF ← array of n buckets, each with m slots, all set to empty
    h ← array of k hash functions
end procedure
procedure add(x)
    i ← 0
    while i < k do
        index ← hi(x)
        if MF[index] has an empty slot then
            Insert x into MF[index]
            return
        end if
        i ← i + 1
    end while
    Perform Morton Filter splitting or resizing
end procedure
procedure Search(x)
    i ← 0
    while i < k do
        index ← hi(x)
        if x is in MF[index] then
            return True
        end if
        i ← i + 1
    end while
    return False
end procedure
procedure Delete(x)
    i ← 0
    while i < k do
        index ← hi(x)
        if x is in MF[index] then
            Remove x from MF[index]
            return
        end if
        i ← i + 1
    end while
    return False
end procedure
```

### 2.5. Vacuum Filter

Bộ lọc Vacuum (Wang et al., 2019) là một cấu trúc dữ liệu xác suất hiệu quả cho phép thêm, xóa và tìm kiếm các phần tử, tương tự như Bộ lọc Cuckoo nhưng hiệu quả hơn về không gian lưu trữ và thông lượng. Bộ lọc Vacuum sử dụng một bảng gồm m bucket, mỗi bucket có nhiều slot (thường là 4) để lưu trữ các mã băm f(x). Các mã băm này có thể được lưu trữ tại một trong hai vị trí được xác định bởi $h1(x) = h(x) \bmod m$ và $h2(x) = h1(x) \oplus (h(f(x)) \bmod m)$. Khác với Bộ lọc Cuckoo, Bộ lọc Vacuum chia các bucket thành các chunk có kích thước bằng nhau L, với L = 2n và m là bội số của L. Khi thêm phần tử x, tính h1(x), h2(x), f(x) và tìm slot trống để lưu f(x). Nếu không có slot trống, di chuyển các mã băm khác để tạo chỗ trống. Khi tìm kiếm x, kiểm tra sự hiện diện của f(x) tại các vị trí xác định. Khi xóa x, tìm và loại bỏ f(x). Bộ lọc Vacuum đạt hiệu quả vượt

trội về không gian và thông lượng so với các bộ lọc Cuckoo, Bloom, Quotient và Morton.

**Thuật toán 5:** Vacuum Filter

```
procedure Initialize(m, n, k)
   VF ← array of n buckets, each with m slots, all set to empty
   h ← array of k hash functions
end procedure

procedure Add(x)
   for i from 0 to k-1 do
      index ← hi(x)
      if VF[index] has an empty slot then
         Insert x into VF[index]
         return
      end if
   end for
   ReorderOrResize()
end procedure

procedure Search(x)
   for i from 0 to k-1 do
      index ← hi(x)
      if x is in VF[index] then
         return True
      end if
   end for
   return False
end procedure

procedure Delete(x)
   for i from 0 to k-1 do
      index ← hi(x)
      if x is in VF[index] then
         Remove x from VF[index]
         return True
      end if
   end for
   return False
end procedure
```

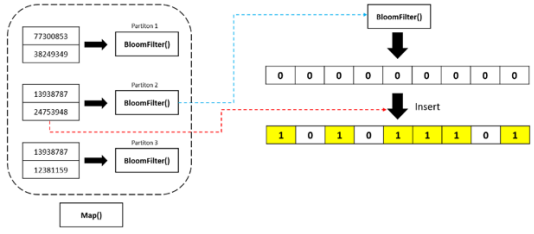
### 2.6. Apache Spark

Apache Spark (Zaharia et al., 2010) là một framework xử lý dữ liệu quy mô lớn có thể thực hiện nhanh chóng việc xử lý tác vụ trên các tập dữ liệu lớn và cũng có thể phân tán trên nhiều máy tính. Spark tương thích với nhiều hệ thống tập tin phân tán như Hadoop HDFS.

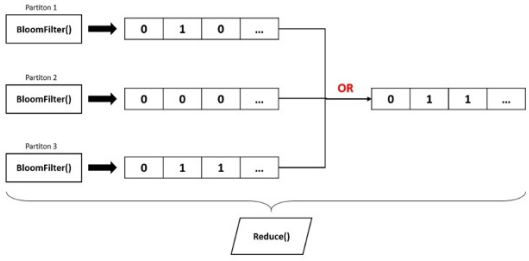
### 2.7. Các bộ lọc trong Apache Spark

#### *2.7.1. Bloom Filter trong Apache Spark*

Bloom Filter là một cấu trúc dữ liệu xác suất được sử dụng rộng rãi trong Apache Spark để kiểm tra xem một phần tử có phải là thành viên của một tập hợp hay không. Nó đặc biệt hữu ích để giảm kích thước tập dữ liệu trước khi thực hiện các hoạt động tốn kém như join. Trong Spark, Bloom Filter được triển khai bằng cách sử dụng một loạt các hàm băm ánh xạ các phần tử vào các vị trí trong một mảng bit, giảm đáng kể nhu cầu về bộ nhớ với chi phí là một tỷ lệ dương tính giả có thể quản lý được. Hình 1 và 2 mô tả hàm Map() và Reduce() khi triển khai Bloom Filter trên Apache Spark.



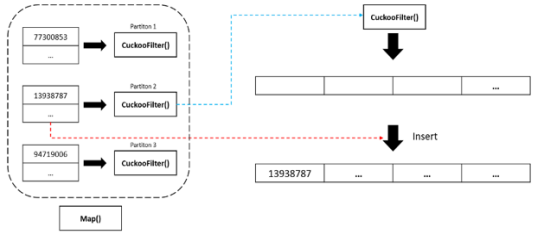

**Hình 1. Hàm Map() khi triển khai Bloom Filter**



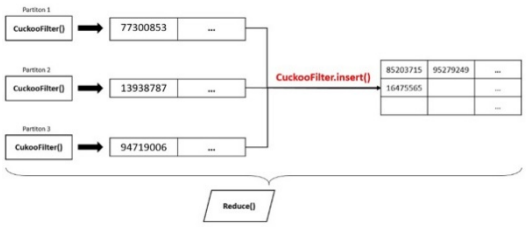

**Hình 2. Hàm Reduce() khi triển khai Bloom Filter**

#### *2.7.2. Cuckoo Filter trong Apache Spark*

Cuckoo Filter mở rộng các khả năng của Bloom Filter bằng cách không chỉ cho phép thêm và truy vấn phần tử mà còn cho phép xóa các phần tử. Điều này làm cho Cuckoo Filter trở nên phù hợp với các tập dữ liệu trong Spark. Nó sử dụng một mảng bucket, trong đó mỗi bucket có thể chứa nhiều mục, cùng với hai hàm băm để xác định vị trí của mục. Hình 3 và 4 thể hiện kiến trúc của Cuckoo Filter trong Apache Spark.



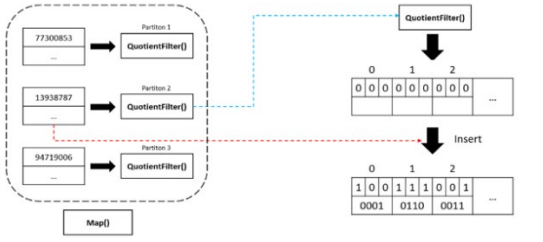

**Hình 3. Hàm Map() khi triển khai Cuckoo Filter**



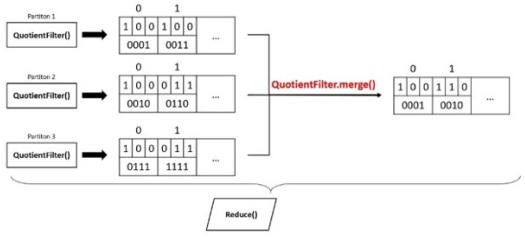

**Hình 4. Hàm Reduce() khi triển khai Cuckoo Filter**

### *2.7.3. Quitient Filter trong Apache Spark*

Quotient Filter là một biến thể của Bloom Filter, tuy nhiên nó sử dụng cách tiếp cận khác để cải thiện hiệu năng. Thay vì sử dụng một mảng bit như Bloom Filter, Quotient Filter lưu trữ các giá trị băm dưới dạng cặp thương số (quotient) và số dư (remainder). Điều này cho phép Quotient Filter nén dữ liệu hiệu quả hơn và giảm không gian lưu trữ so với Bloom Filter truyền thống. Trong Spark, Quotient Filter có thể được triển khai để tối ưu hóa việc lọc và truy vấn trên tập dữ liệu lớn. Việc áp dụng Quotient Filter giúp giảm đáng kể không gian lưu trữ và cải thiện thời gian xử lý. Hình 11 và 12 minh họa kiến trúc của Quotient




**Hình 5. Hàm Map() khi triển khai Quotient Filter**




**Hình 6. Hàm Reduce() khi triển khai Quotient Filter**

### *2.7.4. Morton Filter trong Apache Spark*

Morton Filter là một biến thể mới hơn của các thuật toán lọc xác suất cung cấp cách tiếp cận tối ưu bằng cách kết hợp các kỹ thuật nén và tổ chức bucket để tối ưu hóa cả không gian lưu trữ và hiệu suất. Khi triển khai trong Apache Spark, Morton Filter giảm thiểu việc truy cập bộ nhớ trong quá trình truy vấn, làm cho nó phù hợp với các môi trường đòi hỏi cả hiệu suất cao và sử dụng bộ nhớ hiệu quả. Kiến trúc của Morton Filter trên Apache Spark dựa trên Cuckoo Filter nhưng được cải tiến đáng kể để tăng hiệu suất. Morton Filter sử dụng kỹ thuật nén và tổ chức dưới các bucket giúp tối ưu hóa không gian lưu trữ và giảm số lần truy cập bộ nhớ. Các phần tử trong Morton Filter được băm và lưu trữ trong các bucket, tương tự như Cuckoo Filter, nhưng với cách nén dữ liệu được cải thiện đáng kể. Nhờ vào các cải tiến này, Morton Filter có thể thực hiện các tác vụ tìm kiếm nhanh hơn và hiệu quả hơn trong môi trường xử lý phân tán của Spark

### *2.7.5. Vacuum Filter trong Apache Spark*

Vacuum Filter là một cấu trúc dữ liệu xác suất mới được giới thiệu gần đây như một giải pháp thay thế cho Bloom Filter và Cuckoo Filter. Vacuum Filter kết hợp các kỹ thuật của cả hai bộ lọc này để đạt được hiệu quả tốt hơn về cả không gian lưu trữ và thời gian xử lý. Kiến trúc của Vacuum Filter trên Apache Spark dựa trên Bloom Filter nhưng với các cải tiến để tăng hiệu suất và giảm không gian lưu trữ. Vacuum Filter kết hợp các kỹ thuật từ cả Bloom Filter và Cuckoo Filter, cho phép chèn, xóa và truy vấn các phần tử một cách hiệu quả hơn. Trong Spark, Vacuum Filter được thiết kế để tận dụng tối đa khả năng xử lý song song và phân tán, giúp tối ưu hóa các truy vấn trên tập dữ liệu lớn. Những cải tiến này làm cho Vacuum Filter trở thành một lựa chọn tốt hơn cho các ứng dụng yêu cầu xử lý dữ liệu lớn với hiệu suất cao. Trong Spark, Vacuum Filter có thể được sử dụng để tối ưu hóa các truy vấn trên tập dữ liệu lớn. Nó sử dụng một bảng băm với cấu trúc đặc biệt cho phép chèn, xóa và truy vấn các phần tử một cách hiệu quả. Vacuum Filter cũng hỗ trợ xử lý song song và phân tán trên nhiều nút tính toán trong cụm Spark.

## 3. THỰC NGHIỆM

### 3.1. Môi trường và bộ dữ liệu

Các thực nghiệm được tiến hành trên một cụm 7 máy tính (1 master và 6 nút tính toán). Mỗi máy tính sử dụng hệ điều hành Ubuntu 20.04 LTS và được cấu hình với 4 vCPU, 32GB RAM và 70GB HDD. Môi trường được cài đặt các phần mềm sau: Java 1.8, Hadoop 3.2.2 và Spark 3.2.0. Spark được cấu hình để chạy ở chế độ master với 6 executor, trong đó mỗi executor có 3 CPU và 30GB RAM. HDFS được cấu hình để lưu trữ dữ liệu đầu vào và đầu ra. Các tập dữ liệu được sử dụng để chạy thử nghiệm là

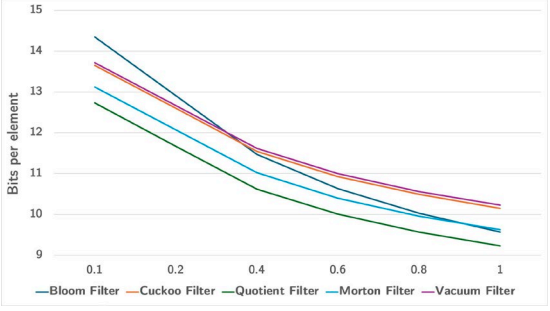
dữ liệu chuẩn được tạo bởi Purdue MapReduce Benchmarks Suite (Ahmad et al., 2012).

Các thử nghiệm thực hiện trên hai môi trường khác nhau: (1) Môi trường local được thực hiện trên một máy tính duy nhất. Apache Spark được cài đặt và cấu hình để chạy trên chế độ local, tận dụng tài nguyên của máy đơn để xử lý dữ liệu; và (2) Môi trường cluster được thực hiện trên một cụm máy tính (cluster) gồm 6 nút tính toán. Apache Spark được cài đặt và cấu hình để chạy trên chế độ cluster, phân phối tính toán trên nhiều nút để xử lý song song dữ liệu. Việc thực hiện thực nghiệm trên cả hai môi trường local và cluster giúp đánh giá toàn diện hiệu năng của các thuật toán lọc trong các điều kiện khác nhau. Môi trường local cho phép kiểm tra khả năng hoạt động và so sánh trực tiếp giữa các thuật toán, trong khi môi trường cluster giúp đánh giá khả năng mở rộng và hiệu quả xử lý phân tán của chúng.

**Bảng 1. Bộ dữ liệu thực nghiệm**

| Test | Dataset L | | Dataset R | |
|---|---|---|---|---|
| | Size | Num_record | Size | Num_record |
| 1 | 2GB | 5,366,662 | 2GB | 2,681,966 |
| 2 | 3GB | 8,049,298 | 3GB | 5,364,698 |

Các tập dữ liệu đều được lưu trữ dưới dạng tệp văn bản thuần túy, mỗi dòng có nhiều hơn một trường được phân tách bằng dấu phẩy. Khóa kết nối là cột đầu tiên của cả hai tập dữ liệu. Số lượng bản ghi và kích thước của các tập dữ liệu được mô tả trong Bảng 1.




**Hình 7. So sánh chi phí lưu trữ (bit)**

Các thử nghiệm được thực hiện với sáu thuật toán bao gồm: Bloom Filter, Cuckoo Filter, Morton Filter, Quotient Filter và Vacuum Filter. Trong thử nghiệm, hai bước được thực hiện để đánh giá độ chính xác của các thuật toán: Đầu tiên áp dụng phép join không có bộ lọc (khoảng cách bằng 0) lên hai tập dữ liệu và so sánh kết quả thu được; sau đó, join có áp dụng các bộ lọc khác nhau lên cùng hai tập dữ liệu đó. Việc so sánh kết quả của hai bước này giúp đảm bảo các thuật toán lọc hoạt động chính xác và cho phép đánh giá hiệu năng của chúng.

### 3.2. Kết quả

Bảng 2 trình bày các công thức tính toán số bit cần thiết để lưu trữ một phần tử trong các bộ lọc khác nhau và kết quả thực nghiệm so sánh chi phí lưu trữ Hình 7 minh họa hiệu suất của một số bộ lọc xác suất, biểu thị chi phí lưu trữ của chúng theo bit trên mỗi phần tử trên một dải tỷ lệ dương tính giả từ 0,1% đến 1%. Ban đầu, Bloom Filter yêu cầu 14,35 bit trên phần tử ở mức tỷ lệ dương tính giả 0,1%, giảm xuống 9,57 bit khi tỷ lệ này tăng lên 1%. Sự giảm dần này minh họa hiệu quả không gian của Bloom Filter ở các ngưỡng lỗi cao hơn. Cuckoo Filter khởi đầu với một lợi thế nhẹ, yêu cầu 13,65 bit cho mỗi phần tử và giảm dần xuống còn 10,15 bit khi tỷ lệ dương tính giả đạt 1%. Đường cong của nó gần như ổn định, thể hiện sự cải thiện khá khiêm tốn về hiệu quả lưu trữ khi chấp nhận tỷ lệ sai sót cao hơn.

Quotient Filter nổi bật với chỉ 12,73 bit ban đầu và giảm xuống còn 9,23 bit khi tỷ lệ dương tính giả đạt 1%. Nó duy trì vị trí dẫn đầu về hiệu quả lưu trữ trong toàn bộ phạm vi khảo sát, khẳng định sự phù hợp của nó cho các ứng dụng nhạy cảm với không gian lưu trữ. Đường cong của Morton Filter bắt đầu ở 13,12 bit và kết thúc ở 9,63 bit, cho thấy một sự cải thiện ổn định nhưng khiêm tốn trong việc sử dụng không gian khi cho phép tỷ lệ lỗi cao hơn. Trong khi đó, Vacuum Filter khởi đầu ở mức 13,72 bit và kết thúc với 10,23 bit tại ngưỡng dương tính giả 1%, thể hiện sự thay đổi ít nhất. Quỹ đạo phẳng của nó cho thấy sự kém hiệu quả hơn trong việc tiết kiệm không gian khi tỷ lệ dương tính giả tăng lên.

**Bảng 2. Công thức tính bit trên phần tử (Fan et al., 2014; Geil et al., 2018; Breslow & Jayasena, 2018; Burdakov et al., 2019; Wang et al., 2019)**

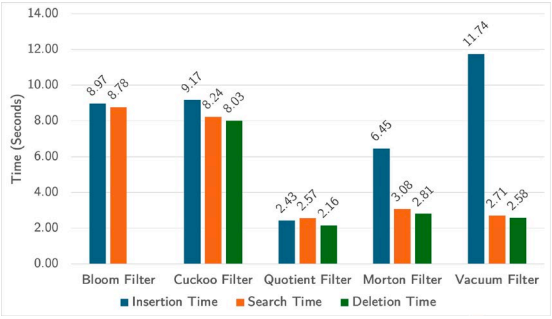
| Filters | Số bit trên phần tử |
|---|---|
| Bloom Filter | $1.44\ log_2(\frac{1}{\varepsilon})$ |
| Cuckoo Filter | $(log_2\left(\frac{1}{\varepsilon}\right) + 3)/\alpha$ |
| Quotient Filter | $(log_2\left(\frac{1}{\varepsilon}\right) + 2.215)/\alpha$ |
| Vacuum Filter | $(log_2\left(\frac{1}{\varepsilon}\right) + 3.07)/\alpha$ |
| Morton Filter | $(log_2\left(\frac{1}{\varepsilon}\right) + 2.5)/\alpha$ |

So sánh thời gian thực thi trong các Hình 8 và 9, hiệu suất của các bộ lọc khác nhau trên ba hoạt động chính là chèn, tìm kiếm và xóa được đánh giá bằng cách sử dụng hai tập dữ liệu có kích thước khác

nhau. Bloom Filter cho thấy thời gian thực thi cân bằng cho các hoạt động chèn và tìm kiếm, lần lượt vào khoảng 8,97 giây và 8,78 giây, không hỗ trợ xóa. Sự cân bằng này biến nó thành một lựa chọn đáng tin cậy cho các tập dữ liệu tĩnh không yêu cầu xóa. Cuckoo Filter có hiệu suất chèn gần tương đồng với Bloom Filter, lần lượt là 9,17 giây và 8,24 giây. Đáng chú ý, thời gian xóa của Cuckoo Filter thấp nhất ở mức 8,02 giây, cho thấy Cuckoo Filter phù hợp hơn cho môi trường có các bản ghi thường xuyên bị xóa. Quotient Filter, mặc dù có thời gian chèn cao nhất là 2,44 giây, nhưng với thời gian tìm kiếm thấp nhất ở mức 2,57 giây. Thời gian xóa cũng tương đối thấp ở mức 2,15 giây. Thời gian tìm kiếm của nó có thể phù hợp cho các ứng dụng ưu tiên hoạt động đọc và chậm hơn cho thao tác cập nhật. Morton Filter có hiệu suất khác biệt đáng kể với thời gian chèn tương đối dài là 6,45 giây nhưng có thời gian tìm kiếm vượt trội là 3,07 giây và thời gian xóa là 2,80 giây. Những đặc điểm này cho thấy Morton Filter có thể được ưu tiên trong các kịch bản mà tìm kiếm nhanh quan trọng hơn chèn. Tuy nhiên, Vacuum Filter có thời gian chèn tăng cao lên 11,73 giây, cao hơn đáng kể so với các bộ lọc khác. Thời gian tìm kiếm và xóa của nó lần lượt là 2,71 giây và 2,58 giây. Thời gian chèn cao có thể là một bất lợi đáng kể cho các ứng dụng yêu cầu cập nhật dữ liệu thường xuyên.

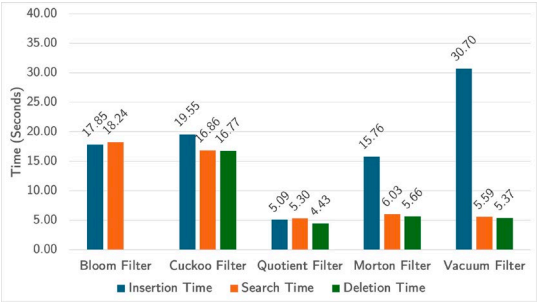
Nhìn chung, khi xem xét thời gian thực thi cho chèn, tìm kiếm và xóa, việc lựa chọn bộ lọc có thể phụ thuộc vào các yêu cầu cụ thể của trường hợp sử dụng. Đối với các tập dữ liệu tĩnh hoặc khi tốc độ chèn không phải là yếu tố quan trọng, Vacuum Filter có thể là một lựa chọn khả thi do thời gian tìm kiếm và xóa thấp của nó. Đối với môi trường có các thao tác cập nhật thường xuyên, hiệu suất cân bằng và thời gian xóa thấp của Cuckoo Filter có thể phù hợp hơn. Ngược lại, khi ưu tiên tốc độ đọc, các bộ lọc Quotient và Morton cung cấp những lợi ích thuyết phục mặc dù thời gian chèn chậm hơn.

So sánh thời gian thực thi trong cụm Spark Bảng 3 và Hình 10 cho thấy kết quả thực nghiệm so sánh thời gian thực thi của các thuật toán lọc trong môi trường cụm Spark đã làm nổi bật những ưu điểm nổi bật của xử lý dữ liệu song song và phân tán. Spark với khả năng thực hiện tính toán phân tán trên nhiều nút, cho phép các thuật toán như Bloom Filter, Cuckoo Filter và đặc biệt là Quotient Filter thể hiện tốc độ xử lý nhanh hơn khi xử lý trên các bộ dữ liệu lớn.

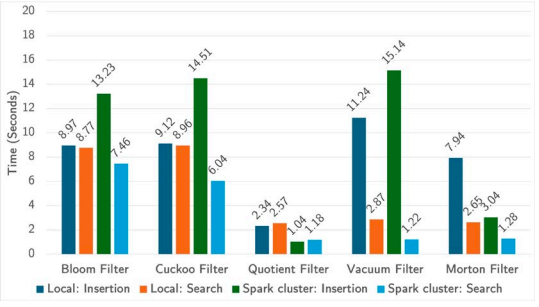



**Hình 8. Tổng thời gian thực thi của các bộ lọc với 3GB dữ liệu (đơn vị: giây)**

Quotient Filter nổi bật với thời gian chèn chỉ 2,34 giây và thời gian tìm kiếm là 2,57 giây, đã thể hiện sự vượt trội so với Bloom Filter (8,97 giây cho chèn, 8,77 giây cho tìm kiếm) và Cuckoo Filter (9,12 giây cho chèn, 8,96 giây cho tìm kiếm). Điều này thể hiện khả năng tận dụng sức mạnh tính toán song song của Spark, cho phép Quotient Filter xử lý nhanh chóng và hiệu quả các bộ dữ liệu lớn, vượt xa hiệu suất của việc chạy tuần tự trên môi trường máy tính đơn.




**Hình 9. Tổng thời gian thực thi của các bộ lọc với 5GB dữ liệu (đơn vị: giây)**




**Hình 10. So sánh thời gian xử lý trên môi trường cục bộ và phân tán trên Spark**

Hình 10 cho thấy sự cải thiện đáng kể về hiệu năng của các thuật toán khi chuyển từ môi trường local sang cluster, đặc biệt là với Quotient Filter. Kết quả cho thấy Quotient Filter đạt được mức cải thiện hơn 50% khi chạy trên cluster so với local, vượt trội

hơn hẳn so với các thuật toán khác. Điều này có thể được giải thích bởi kiến trúc phù hợp và dễ dàng triển khai với xử lý song song, Quotient Filter có thể tận dụng hiệu quả các tài nguyên phần cứng và băng thông mạng của cluster, dẫn đến sự cải thiện đáng kể về hiệu năng xử lý.

**Bảng 3. Tổng thời gian thực hiện bộ lọc với 2gb dữ liệu (đơn vị: giây)**

| | Local | | Spark cluster | |
|---|---|---|---|---|
| | **Insertion time** | **Search time** | **Insertion time** | **Search time** |
| **Bloom Filter** | 8,97 | 8,77 | 13,23 | 7,46 |
| **Cuckoo Filter** | 9,12 | 8,96 | 14,51 | 6,04 |
| **Quotient Filter** | 2,34 | 2,57 | 1,04 | 1,18 |
| **Morton Filter** | 11,24 | 2,87 | 15,14 | 1,22 |
| **Vacuum Filter** | 7,94 | 2,65 | 3,04 | 1,28 |

Đáng lưu ý là khi được triển khai trên Spark, tất cả các thuật toán này đều hưởng lợi từ tính song song dữ liệu và xử lý song song trên nhiều nút tính toán. Tuy nhiên, mức độ cải thiện hiệu suất khác nhau tùy thuộc vào đặc điểm và điểm mạnh của từng thuật toán. Quotient Filter với những tối ưu hóa về không gian lưu trữ và thời gian thực thi đã chứng minh là rất phù hợp và hiệu quả khi kết hợp với sức mạnh của Spark. Kết quả cho thấy việc tận dụng xử lý song song và phân tán trên các nền tảng như Spark mang lại những lợi ích đáng kể cho các ứng dụng xử lý dữ liệu quy mô lớn. Nó không chỉ giảm đáng kể thời gian tính toán mà còn mở ra khả năng mở rộng và xử lý các bộ dữ liệu có kích thước lớn. Phương pháp tiếp cận này là một xu hướng đầy hứa hẹn và ngày càng phổ biến trong lĩnh vực khoa học dữ liệu và xử lý thông tin hiện đại.

Kết quả của nghiên cứu này nhìn chung phù hợp với những công trình liên quan trước đây. Pandey et al. (2017) cũng chỉ ra rằng Quotient Filter có hiệu quả vượt trội về không gian lưu trữ và thời gian thực thi khi so sánh với Bloom Filter và Cuckoo Filter. Tương tự, Agarwal et al. (2013) nhận thấy sự cải thiện đáng kể về hiệu năng của các bộ lọc xác suất, đặc biệt là Quotient Filter, khi triển khai trên Apache Spark. Những kết quả này củng cố thêm tính đúng đắn và ý nghĩa của các phát hiện trong nghiên cứu. Tuy nhiên, việc có thêm nhiều nghiên cứu với các tập dữ liệu và kịch bản đa dạng hơn là cần thiết để đánh giá toàn diện hiệu năng của các thuật toán lọc, đặc biệt là với sự xuất hiện của các biến thể mới như Morton Filter và Vacuum Filter.

## 4. KẾT LUẬN

Trong nghiên cứu này, các thuật toán lọc hiệu quả trong môi trường xử lý dữ liệu lớn như Bloom Filter, Cuckoo Filter, Quotient Filter, Morton Filter và Vacuum Filter được trình bày chi tiết và đánh giá hiệu suất. Kết quả thực nghiệm cho thấy Quotient Filter là thuật toán hiệu quả nhất về mặt lưu trữ và cân bằng tốt giữa các tiêu chí hiệu năng. Quotient Filter sử dụng ít không gian lưu trữ nhất cho mỗi phần tử và đạt tốc độ chèn, tìm kiếm, xóa ở mức cao và ổn định. Bên cạnh đó, các thuật toán khác cũng thể hiện những ưu điểm riêng phù hợp với các yêu cầu cụ thể. Bloom Filter có tốc độ tốt, thích hợp cho các ứng dụng yêu cầu chèn và tìm kiếm nhanh trên dữ liệu tĩnh. Morton Filter nổi trội về tốc độ tìm kiếm nhưng chậm hơn khi chèn dữ liệu mới, phù hợp khi ưu tiên tốc độ đọc. Vacuum Filter, mặc dù có thời gian chèn chậm nhất, nhưng cung cấp khả năng tìm kiếm và xóa nhanh, là một lựa chọn tốt cho các trường hợp yêu cầu tìm kiếm và xóa nhanh mà không cần chèn dữ liệu thường xuyên. Khi được triển khai trên nền tảng Spark với khả năng xử lý song song và phân tán, tốc độ xử lý của các thuật toán được cải thiện đáng kể. Đặc biệt, Quotient Filter đạt mức cải thiện vượt bậc, hơn 50% khi chạy trên cluster so với môi trường local. Điều này mở ra triển vọng ứng dụng hiệu quả các thuật toán lọc, đặc biệt là Quotient Filter, để xử lý dữ liệu lớn và phân tán.

## LỜI CẢM TẠ

Nghiên cứu này được thực hiện với sự tài trợ của Trường Đại học Cần Thơ (Đề tài T2024-88).

## TÀI LIỆU THAM KHẢO (REFERENCES)

Agarwal, S., Mozafari, B., Panda, A., Milner, H., Madden, S., & Stoica, I. (2013). BlinkDB: Queries with bounded errors and bounded response times on very large data. *Proceedings of the 8th ACM European Conference on Computer Systems, EuroSys 2013*, 29–42. ACM. https://doi.org/10.1145/2465351.2465355

Ahmad, F., Lee, S., Thottethodi, M., & Vijaykumar, T. N. (2012). PUMA: Purdue MapReduce Benchmarks Suite. In *ECE Technical Reports*.

Al-hisnawi, M., & Ahmadi, M. (2016). Deep Packet Inspection Using Quotient Filter. *IEEE Communications Letters*, *20*(11), 2217–2220. https://doi.org/10.1109/LCOMM.2016.2601898

Breslow, A. D., & Jayasena, N. S. (2018). Morton filters: Faster, spaceefficient cuckoo filters via biasing, compression, and decoupled logical sparsity. *Proceedings of the VLDB Endowment*, *11*(9). VLDB Endowment. https://doi.org/10.14778/3213880.3213884

Burdakov, A., Ermakov, E., Panichkina, A., Ploutenko, A., Grigorev, U., Ermakov, O., & Proletarskaya, V. (2019). Bloom Filter Cascade Application to SQL Query Implementation on Spark. *Proceedings - 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, PDP 2019*. IEEE. https://doi.org/10.1109/EMPDP.2019.8671557

Chaudhuri, S., Ganti, V., & Kaushik, R. (2006). A primitive operator for similarity joins in data cleaning. *Proceedings - International Conference on Data Engineering*. IEEE. https://doi.org/10.1109/ICDE.2006.9

Ezzaki, F., Abghour, N., Elomri, A., Moussaid, K., & Rida, M. (2020). Bloom filter and its variants for the optimization of MapReduce's algorithms: A review. *Proceedings of 2020 5th International Conference on Cloud Computing and Artificial Intelligence: Technologies and Applications, CloudTech 2020*. IEEE. https://doi.org/10.1109/CloudTech49835.2020.9365876

Fan, B., Andersen, D. G., Kaminsky, M., & Mitzenmacher, M. D. (2014). Cuckoo filter: Practically better than bloom. *CoNEXT 2014 - Proceedings of the 2014 Conference on Emerging Networking Experiments and Technologies*. Association for Computing Machinery. https://doi.org/10.1145/2674005.2674994

Fier, F., Augsten, N., Bouros, P., Leser, U., & Freytag, J. C. (2018). Set similarity joins on MapReduce: An experimental survey. *Proceedings of the VLDB Endowment*, *11*(10). VLDB Endowment. https://doi.org/10.14778/3231751.3231760

García, S., Ramírez-Gallego, S., Luengo, J., Benítez, J. M., & Herrera, F. (2016). Big data preprocessing: methods and prospects. *Big Data Analytics*, *1*(1). https://doi.org/10.1186/s41044-016-0014-0

Geil, A., Farach-Colton, M., & Owens, J. D. (2018). Quotient Filters: Approximate Membership Queries on the GPU. *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 451–462. IEEE. https://api.semanticscholar.org/CorpusID:3991218

Kumar, N., Sai, K. H. S., Hordiichuk, V., Menon, R., Catherene, J. A. C., Saha, G. C., & Balaji, K., (2023). Harnessing the Power of Big Data: Challenges and Opportunities in Analytics. *Tuijin Jishu/Journal of Propulsion Technology*, *44*(2). https://doi.org/10.52783/tjjpt.v44.i2.193

Li, L. (2021). Efficient Distributed Database Clustering Algorithm for Big Data Processing. *Proceedings - 2021 6th International Conference on Smart Grid and Electrical Automation, ICSGEA 2021*. IEEE. https://doi.org/10.1109/ICSGEA53208.2021.00118

Lu, Y., Prabhakar, B., & Bonomi, F. (2005). Bloom filters: Design innovations and novel applications. *43rd Annual Allerton Conference on Communication, Control and Computing 2005*, *2*.

Maulana, M. S., Linuwih, B. P., Nuha, H. H., & Satrya, G. B. (2023, August). Bloom, Xor, and Cuckoo Filter Comparison for Database's Query Optimization. *International Conference on ICT Convergence*. IEEE. https://doi.org/10.1109/ICoICT58202.2023.10262536

Pandey, P., Bender, M. A., Johnson, R., & Patro, R. (2017). A general-purpose counting filter: Making every bit count. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, *Part F127746*, 775–787. ACM. https://doi.org/10.1145/3035918.3035963

Tran, T. T. Q., Phan, T. C., Laurent, A., & D'Orazio, L. (2020, July). Optimization for large-scale fuzzy joins using fuzzy filters in MapReduce. *International Conference on Fuzzy Systems* (FUZZ-IEEE) (pp 1-8). IEEE. . https://doi.org/10.1109/FUZZ48607.2020.9177610

Wang, M., Zhou, M., Shi, S., & Qian, C. (2019). Vacuum filters: More space-efficient and faster replacement for bloom and cuckoo filters. *Proceedings of the VLDB Endowment*, *13*(2). VLDB Endowment. https://doi.org/10.14778/3364324.3364333

Yan, C., Zhao, X., Zhang, Q., & Huang, Y. (2017). Efficient string similarity join in multi-core and distributed systems. *PLoS ONE*, *12*(3). https://doi.org/10.1371/journal.pone.0172526

Yu, M., Li, G., Deng, D., & Feng, J. (2016). String similarity search and join: a survey. In *Frontiers of Computer Science* (Vol. 10, Issue 3). https://doi.org/10.1007/s11704-015-5900-5

Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., & Stoica, I. (2010). Spark: cluster computing with working sets. *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, 10. USENIX Association. https://doi.org/ 10.5555/1863103.1863113.