🤔 Lựa Chọn Nền Tảng Cho Customer Identity Resolution (CIR): PostgreSQL 16 Hay OpenSearch?

Customer Identity Resolution không chỉ đơn thuần là tìm kiếm. Đó là một quy trình phức tạp yêu cầu **logic nghiệp vụ chặt chẽ, khả năng xử lý giao dịch, đảm bảo tính toàn vẹn dữ liệu** và khả năng mở rộng có kiểm soát. Khi đối chiếu PostgreSQL 16 (với các extension hiện đại) và OpenSearch, PostgreSQL nổi lên như một ứng cử viên sáng giá và thường phù hợp hơn cho vai trò là **trái tim xử lý** của hệ thống CIR.

✨ 9 Lý Do Nên Chọn PostgreSQL 16 Làm Nền Tảng Chính Cho CIR

Dưới đây là các lý do chi tiết, kèm theo ví dụ cụ thể để bạn dễ hình dung:

---

1. Xử Lý Logic Ghép Nối Phức Tạp Tinh Gọn Với Stored Procedures

- **Điểm mạnh:** PostgreSQL cho phép bạn đóng gói toàn bộ logic xử lý CIR (kiểm tra dữ liệu mới, áp dụng rule fuzzy matching, validate, merge profile, cập nhật master ID...) vào trong một hoặc một vài Stored Procedures.
- **Vi sao (Why) ?** Thay vì phải dàn trải logic qua nhiều hàm Lambda rời rạc, khó quản lý và dễ gây lỗi liên lạc, Stored Procedure giữ mọi thứ **tập trung, dễ debug và sửa đổi**. Quan trọng nhất, chúng chạy **atomic** – toàn bộ quy trình merge hoặc thành công hoặc thất bại hoàn toàn, đảm bảo tính toàn vẹn.
- **Ví dụ:** Hãy tưởng tượng quy trình `resolve_customer_identities(new_profile_id)`. Procedure này có thể:
    1. Tìm các profile có thể liên quan dựa trên email, số điện thoại chính xác.
    2. Áp dụng rule fuzzy matching cho tên, địa chỉ (dùng các hàm text similarity).
    3. Kiểm tra các ràng buộc nghiệp vụ (ví dụ: không merge nếu cùng một thiết bị nhưng khác quá nhiều thông tin).
    4. Nếu tìm thấy profile master phù hợp: liên kết profile mới với master đó.
    5. Nếu tìm thấy nhiều profile master có thể là một: chạy logic gộp (merge), chọn ra master chính và đánh dấu các profile còn lại là alias.
    6. Nếu không tìm thấy: tạo một profile master mới.
    - Thực hiện tất cả các bước này trong **một giao dịch duy nhất** là cực kỳ mạnh mẽ và an toàn so với việc gọi nhiều services bên ngoài.

PROF

---

2. Phản Ứng Tức Thời Với Dữ Liệu Mới Bằng Event Triggers

- **Điểm mạnh:** Triggers trong PostgreSQL cho phép tự động kích hoạt Stored Procedure (hoặc hàm) ngay lập tức sau khi có sự kiện `INSERT`, `UPDATE`, `DELETE` trên bảng dữ liệu đầu vào (bảng staging hoặc bảng profile).
- **Vi sao (Why) ?** Điều này tạo ra một luồng xử lý **gần như real-time**, không cần đến cơ chế polling tốn kém hoặc phải phụ thuộc vào các hệ thống queue/stream phức tạp chỉ để báo hiệu có dữ liệu mới. Luồng xử lý CIR được bắt đầu **ngay tại nguồn dữ liệu**.
- **Ví dụ:**

- Bạn có bảng `customer_staging` chứa dữ liệu mới từ các nguồn khác nhau.
- Tạo `AFTER INSERT OR UPDATE ON customer_staging EXECUTE PROCEDURE trigger_resolve_new_profile();`
- Ngay khi một dòng mới được thêm vào `customer_staging`, trigger sẽ gọi hàm `trigger_resolve_new_profile`, hàm này sau đó gọi Stored Procedure xử lý CIR cho bản ghi mới đó. Đơn giản và hiệu quả!

---

3. Chi Phí Vận Hành Thấp Hơn Đáng Kể

- **Điểm mạnh:** Chi phí cho PostgreSQL thường thấp hơn OpenSearch, đặc biệt khi tính đến nhu cầu xử lý logic phức tạp.
- **Vi sao (Why) ?** OpenSearch được tối ưu cho tìm kiếm, đòi hỏi tài nguyên (CPU, RAM, IOPS) cho việc indexing và duy trì các shard/replica. Xử lý logic phức tạp trên OpenSearch thường yêu cầu scale-out (thêm node), làm tăng chi phí đáng kể. PostgreSQL, đặc biệt là các instance được tối ưu cho compute, có thể xử lý logic nặng hiệu quả hơn trên một hoặc ít instance hơn.
- **Ví dụ:**
  - Một instance RDS PostgreSQL `r6g.xlarge` (4 vCPU, 32GB RAM) có thể xử lý lượng lớn logic CIR với chi phí khoảng **$170–220/tháng** (ước tính cơ bản).
  - Một cụm OpenSearch tối thiểu cho production (ví dụ: 2 nodes `i3.large` hoặc tương đương) có thể tốn từ **$300–500/tháng** trở lên, và chi phí này tăng nhanh khi bạn cần nhiều tài nguyên hơn cho indexing hoặc query phức tạp (không phải search đơn thuần).
  - Việc dàn trải logic sang Lambda cũng cộng thêm chi phí tính theo Request và Duration, có thể trở nên rất đắt đỏ với lượng dữ liệu lớn.

---

4. Tích Hợp Vector Search Mạnh Mẽ Với `pgvector`

- **Điểm mạnh:** Extension `pgvector` cho phép bạn lưu trữ và tìm kiếm các vector embedding trực tiếp trong PostgreSQL, hỗ trợ các thuật toán tìm kiếm láng giềng gần nhất (KNN).
- **Vi sao (Why) ?** Bạn có thể kết hợp dữ liệu có cấu trúc truyền thống (email, số điện thoại) với các tín hiệu hiện đại như embedding của tên, địa chỉ, mô tả hành vi hoặc nội dung tương tác để ghép profile. Tất cả diễn ra **trong cùng một cơ sở dữ liệu**, đơn giản hóa kiến trúc và logic.
- **Ví dụ:**
  - Bạn có thể tạo embedding cho tên khách hàng (`customer_name_embedding`) và địa chỉ (`address_embedding`).
  - Trong Stored Procedure CIR, bạn có thể tìm kiếm các profile tiềm năng bằng SQL như:

```
SELECT profile_id
FROM profiles
WHERE email = '...' OR phone = '...'
   OR (
        customer_name_embedding <=> ? < 0.1  -- fuzzy match tên
qua vector similarity
       AND address_embedding <=> ? < 0.1      -- fuzzy match
địa chỉ qua vector similarity
      );
```

- Kết hợp logic này với các rule dựa trên dữ liệu có cấu trúc truyền thống là điều PostgreSQL làm rất tốt.

---

5. Đảm Bảo Tính Toàn Vẹn Dữ Liệu Với Giao Dịch & ACID

- **Điểm mạnh:** PostgreSQL tuân thủ nghiêm ngặt các thuộc tính ACID (Atomicity, Consistency, Isolation, Durability), đặc biệt với các giao dịch.
- **Vi sao (Why) ?** Trong CIR, việc gộp hai profile hoặc liên kết một profile mới là những thao tác **quan trọng và nhạy cảm**. Bạn cần đảm bảo rằng các thay đổi này (cập nhật master ID, đánh dấu profile cũ, thêm bản ghi lịch sử) hoặc thành công hoàn toàn hoặc không có gì xảy ra (rollback). OpenSearch với mô hình nhất quán cuối cùng (eventual consistency) không thể cung cấp mức độ đảm bảo này, dễ dẫn đến các trường hợp duplicate master hoặc liên kết sai trong các tình huống cạnh tranh (race conditions).
- **Ví dụ:** Hai event cùng lúc báo cáo hoạt động từ cùng một người dùng nhưng với profile ID tạm thời khác nhau.
    - **PostgreSQL:** Stored Procedure chạy trong một giao dịch. Nếu cả hai cùng cố gắng tạo master hoặc gộp vào cùng một master, hệ thống transaction sẽ xử lý các xung đột khóa một cách an toàn. Một trong hai giao dịch có thể phải đợi hoặc rollback, nhưng dữ liệu cuối cùng sẽ nhất quán và không có duplicate master không mong muốn do race condition.
    - **OpenSearch + Lambda:** Hai hàm Lambda xử lý hai event có thể chạy song song. Do OpenSearch nhất quán cuối cùng, cả hai Lambda có thể đọc trạng thái cũ (chưa có master), dẫn đến việc cả hai cùng tạo ra một master ID mới hoặc gộp sai. Kết quả là duplicate master hoặc trạng thái dữ liệu không chính xác, rất khó khắc phục.

---

6. Dữ Liệu "Sạch Từ Đầu" Nhờ Schema Rõ Ràng & Ràng Buộc

- **Điểm mạnh:** PostgreSQL cho phép định nghĩa schema chặt chẽ với các ràng buộc (`NOT NULL`, `UNIQUE`, `CHECK`, Foreign Key...).
- **Vi sao (Why) ?** Bạn có thể áp đặt các rule về chất lượng dữ liệu ngay tại lớp database. Điều này giúp **giảm thiểu dữ liệu bẩn** đi vào hệ thống CIR ngay từ đầu, đơn giản hóa logic xử lý và tăng độ tin cậy của kết quả resolution.
- **Ví dụ:**
    - Bảng `profiles` yêu cầu `email` là `UNIQUE` và `NOT NULL` (nếu có).
    - Bảng `profile_links` có foreign key tới bảng `profiles` để đảm bảo mọi liên kết đều trỏ tới một profile master thực tế.
    - Sử dụng kiểu dữ liệu `ENUM` cho các trạng thái profile (ví dụ: 'master', 'alias', 'pending_merge').
    - OpenSearch là schema-less (hoặc schema-on-read ở mức lỏng lẻo hơn) → dễ dàng thêm các bản ghi có cấu trúc hoặc kiểu dữ liệu không nhất quán, đẩy gánh nặng làm sạch dữ liệu về phía ứng dụng hoặc các pipeline ETL/ELT phức tạp sau này.

PROF

---

7. Kiểm Soát Xử Lý Batch Theo Tài Nguyên

- **Điểm mạnh:** PostgreSQL cung cấp các công cụ mạnh mẽ để xử lý dữ liệu theo lô (batch) với khả năng kiểm soát tài nguyên sử dụng (ví dụ: `LIMIT`, `OFFSET`, cursor, cấu hình `work_mem`).
- **Vi sao (Why) ?** Khi cần xử lý hàng triệu profile lịch sử hoặc chạy lại quy trình CIR cho một tập dữ liệu lớn, bạn cần khả năng chia nhỏ công việc và xử lý từng phần một cách hiệu quả mà không làm sập hệ thống. PostgreSQL cho phép bạn viết logic batch có ý thức về bộ nhớ và thời gian thực thi.
- **Ví dụ:** Bạn có thể viết một Stored Procedure để:
    1. Chọn 1000 profile `WHERE status = 'unresolved'` sử dụng `LIMIT` và `OFFSET`.
    2. Vòng lặp qua 1000 profile này, gọi logic resolve cho từng profile (hoặc nhóm nhỏ hơn).
    3. Commit thay đổi sau mỗi 1000 profile.
    4. Sử dụng Cursor để xử lý các tập dữ liệu cực lớn mà không tải toàn bộ vào bộ nhớ.
    - Bạn có thể điều chỉnh `work_mem` của session để tối ưu hiệu năng join hoặc sort trong các batch lớn. Điều này cung cấp sự kiểm soát mà việc chỉ dựa vào các hàm Lambda với giới hạn timeout và memory cố định không thể có được.

---

8. Truy Vấn Dữ Liệu Đã Resolve Trực Tiếp Bằng SQL

- **Điểm mạnh:** Sau khi quá trình CIR hoàn tất và các profile đã được liên kết với master ID, dữ liệu kết quả nằm ngay trong PostgreSQL và sẵn sàng để truy vấn bằng SQL tiêu chuẩn.
- **Vi sao (Why) ?** Không cần phải xây dựng thêm các pipeline ETL/sync phức tạp chỉ để đưa dữ liệu đã xử lý sang một hệ thống khác (như OpenSearch chỉ để truy vấn cơ bản). Các đội phân tích, marketing hoặc công cụ BI có thể kết nối trực tiếp để lấy thông tin về profile master, các profile liên kết, lịch sử merge, v.v.
- **Ví dụ:**
    - "Show tôi danh sách các master profile được tạo trong tháng này cùng với số lượng profile alias của mỗi master."
    - "Liệt kê các master profile có cả email và số điện thoại đã được xác thực."
    - "Tính tổng giá trị đơn hàng của tất cả các profile liên kết với master ID X."
    - Tất cả những truy vấn này rất dễ dàng và hiệu quả trên PostgreSQL.

PROF

9. Hệ Sinh Thái Mở Rộng & Cộng Đồng Lớn

- **Điểm mạnh:** PostgreSQL là cơ sở dữ liệu mã nguồn mở với lịch sử lâu đời, sự ổn định đã được kiểm chứng và một cộng đồng phát triển, hỗ trợ cực kỳ lớn.
- **Vi sao (Why) ?** Điều này có nghĩa là bạn dễ dàng tìm thấy tài liệu, công cụ, extension hữu ích (như `pgvector`, `pg_partman` để quản lý phân vùng, `pg_cron` để lên lịch chạy procedure, các công cụ monitoring như `pg_stat_statements`). Khả năng tích hợp với các công nghệ khác (Python, Java, Node.js, Kafka, Airflow, BI tools) là rất mạnh mẽ. Bạn **không bị khóa chặt** vào một nhà cung cấp dịch vụ duy nhất (như OpenSearch Service của AWS), có thể tự host, chuyển sang các nhà cung cấp cloud khác hoặc nâng cấp version dễ dàng.

⚠️ Rủi Ro Nghiêm Trọng Khi Dùng OpenSearch + Lambda Cho Xử Lý Chính CIR

Mặc dù OpenSearch tuyệt vời cho việc tìm kiếm, việc dựa vào nó và Lambda cho logic xử lý CIR cốt lõi mang lại nhiều rủi ro lớn:

❌ 1. Bản Chất Không Phù Hợp Cho Logic & JOIN Phức Tạp

- **Vấn đề:** OpenSearch là search engine, không phải relational database. Nó không được thiết kế để thực hiện các thao tác JOIN phức tạp trên nhiều bảng hoặc áp dụng các rule logic nhiều bước, có điều kiện dựa trên trạng thái dữ liệu hiện tại.
- **Rủi ro:** Logic CIR phải được viết hoàn toàn ở lớp ứng dụng (Lambda). Điều này dẫn đến code dàn trải, khó quản lý, khó đảm bảo tính nhất quán giữa các bước và khó debug khi có lỗi xảy ra trong luồng xử lý kéo dài.

❌ 2. Giới Hạn Concurrency Của Lambda – Nguy Cơ Mất Dữ Liệu Hoặc Tắc Nghẽn

- **Vấn đề:** Mỗi tài khoản AWS có giới hạn về số lượng hàm Lambda có thể chạy đồng thời (mặc định thường là 1000).
- **Rủi ro:** Nếu hệ thống CIR nhận được lượng dữ liệu mới đột biến (ví dụ: từ chiến dịch marketing lớn, traffic tăng đột ngột) khiến số lượng trigger Lambda vượt quá giới hạn này, các yêu cầu xử lý mới sẽ bị **throttle (bị từ chối)**. Điều này có thể dẫn đến **mất dữ liệu vĩnh viễn** (nếu không có cơ chế retry/DLQ vững chắc) hoặc tạo ra độ trễ và tắc nghẽn lớn trong pipeline xử lý CIR, khiến dữ liệu master không được cập nhật kịp thời.

❌ 3. Giới Hạn Thời Gian Chạy (Timeout) & Bộ Nhớ Của Lambda

- **Vấn đề:** Hàm Lambda có giới hạn thời gian chạy tối đa (hiện tại là 15 phút) và giới hạn bộ nhớ (tối đa 10GB).
- **Rủi ro:** Logic CIR, đặc biệt là các bước fuzzy matching trên lượng lớn dữ liệu hoặc xử lý các profile có lịch sử phức tạp, có thể tốn nhiều CPU và RAM. Các hàm Lambda xử lý logic nặng dễ bị **timeout** hoặc **hết bộ nhớ**, dẫn đến việc một số bản ghi không bao giờ được resolve hoặc merge đúng cách. Xử lý batch lớn trong Lambda cũng trở nên khó khăn và rủi ro.

PROF

❌ 4. Chi Phí Indexing Cao & Độ Trễ Indexing

- **Vấn đề:** OpenSearch yêu cầu indexing dữ liệu trước khi có thể tìm kiếm. Quá trình indexing này tốn tài nguyên (CPU, RAM, IOPS) và có chi phí không nhỏ, đặc biệt với lượng dữ liệu thay đổi liên tục.
- **Rủi ro:** Có một độ trễ nhất định từ lúc dữ liệu được ghi vào OpenSearch đến lúc nó thực sự searchable (thường vài giây hoặc hơn tùy tải). Điều này có nghĩa là hệ thống CIR dựa trên OpenSearch không thể đạt được trạng thái "thực sự real-time" khi logic phụ thuộc vào việc tìm kiếm dữ liệu vừa được ghi vào.

❌ 5. Nhất Quán Cuối Cùng (Eventual Consistency) – Gây Lỗi Khi Gộp Profile

- **Vấn đề:** OpenSearch hoạt động theo mô hình nhất quán cuối cùng. Thay đổi trên một shard có thể mất một lúc để lan truyền và hiển thị trên các shard khác hoặc khi truy vấn toàn bộ index.

- **Rủi ro:** Đây là rủi ro **nghiêm trọng nhất** đối với logic gộp profile trong CIR. Nếu hai quy trình (ví dụ: hai hàm Lambda) cùng lúc xử lý hai profile A và B được phát hiện là của cùng một người, và chúng cố gắng gộp hoặc liên kết các profile này trong một hệ thống nhất quán cuối cùng, có khả năng xảy ra race condition. Cả hai quy trình có thể đọc trạng thái cũ, dẫn đến việc tạo ra duplicate master record không mong muốn hoặc các liên kết profile bị sai lệch, làm hỏng dữ liệu master.

❌ 6. Khó Debug & Giám Sát

- **Vấn đề:** Logic xử lý CIR nằm rải rác trên nhiều hàm Lambda và tương tác với OpenSearch thông qua API.
- **Rủi ro:** Khi có lỗi xảy ra (ví dụ: một profile không được resolve đúng, một trường hợp duplicate master), việc truy vết nguyên nhân trở nên rất khó khăn. Bạn phải kiểm tra log của nhiều hàm Lambda khác nhau, theo dõi request/response đến OpenSearch, và cố gắng tái hiện lại luồng xử lý phân tán. So với việc debug một Stored Procedure trong PostgreSQL với các công cụ và log tập trung, đây là một cơn ác mộng.

✅ Gợi Ý Kiến Trúc Hybrid Tối Ưu

Một kiến trúc hiệu quả và thực tế cho CIR là **sử dụng thế mạnh của cả hai hệ thống**:

```
graph LR
    A[Data Sources<br>(Website, App, CRM, etc.)] --> B(Staging
Table<br>PostgreSQL)
    B -- INSERT/UPDATE --> C{PostgreSQL Trigger}
    C --> D(PostgreSQL Stored Procedure<br>Identity Resolution Logic)
    D -- Reads/Writes --> E(PostgreSQL<br>Profiles & Master Data)
    E -- Sync (e.g., Debezium, ETL)<br> Resolved Data -->
F(OpenSearch<br>For Search/Analytics)
    F --> G(Applications / BI Tools<br>For Search & Reporting)
    E --> H(Applications / BI Tools<br>For Direct Data Access/Reporting)
```

PROF

Quy trình:

1. Dữ liệu mới từ các nguồn được đưa vào bảng staging trong **PostgreSQL**.
2. `TRIGGER` trong PostgreSQL tự động kích hoạt `Stored Procedure` xử lý CIR.
3. `Stored Procedure` thực hiện toàn bộ logic complex (fuzzy matching, rule, merge...) trực tiếp trên dữ liệu trong **PostgreSQL**, đảm bảo ACID và tính toàn vẹn. Kết quả (master ID, liên kết profile) được lưu trữ ngay trong PostgreSQL.
4. **PostgreSQL** trở thành nguồn dữ liệu **chính xác và đáng tin cậy** cho các master profile đã resolve.
5. Dữ liệu master profile đã resolve từ PostgreSQL được **đồng bộ (sync)** sang **OpenSearch** (có thể dùng các công cụ như Debezium để bắt thay đổi - CDC, hoặc ETL job đơn giản).
6. **OpenSearch** được sử dụng cho các tác vụ **tối ưu cho tìm kiếm** như tìm kiếm profile theo tên/địa chỉ tự do, autocomplete trên UI, hoặc các dashboard phân tích cần search nhanh.
7. Các ứng dụng hoặc công cụ BI có thể truy cập dữ liệu **master đã resolve** trực tiếp từ **PostgreSQL** (khi cần dữ liệu chính xác, transaction, hoặc JOIN phức tạp) hoặc từ **OpenSearch** (khi cần tìm

kiếm nhanh).

Kiến trúc này cho phép bạn tận dụng khả năng xử lý logic mạnh mẽ, đảm bảo dữ liệu của PostgreSQL làm trái tim CIR, đồng thời vẫn sử dụng tốc độ tìm kiếm của OpenSearch cho lớp hiển thị và phân tích.
