Created by- Neeti Ishan Bhatt

# Lead Deduplication: Approaches Comparison and Analysis

## Introduction: Deduplication Rules and Context

When working with a dataset of leads, duplicates need to be reconciled based on **certain rules**:

1. **Duplicate Identification**:
   - Leads with the same _id are duplicates.
   - Leads with the same email (case-insensitive) are also duplicates.
   - Other fields don't determine duplication.
2. **Resolution Strategy**:
   - The lead with the **latest** entryDate is preferred.
   - If two leads share the **same entryDate**, the **last occurrence** in the list is preferred.

## Approach 1: Simple Map-Based Deduplication

Approach 1 uses two hash maps to detect duplicates during a single pass:

- idMap keyed by _id
- emailToIdMap keyed by normalized email

For each lead:

- If the _id or email is missing, it is classified as a bad lead.
- Check if the _id or email exists in the maps.
- If found, the existing and new lead are compared using a preference strategy (e.g., latest entryDate wins).
- The preferred lead replaces the existing one, maps updated accordingly.
- Field-level changes are logged.
- Otherwise, the lead is added as new.

This approach handles duplicates based on the presence of _id or email but **does not resolve transitive duplicates** where two leads might be linked through intermediate records (e.g., Lead A shares email with Lead B, Lead B shares _id with Lead C).

**<u>Algorithm:</u>**

1. Initialize empty idMap, emailToIdMap, badLeads list.
2. For each lead:
   - Validate presence of _id and email; if missing, add to badLeads.
   - Normalize email.
   - Check for existing lead by _id or email.
   - If found, select preferred lead (using strategy), update maps.
   - Log changes if any differences detected.
   - If no existing lead found, add to maps.
3. Return deduplicated leads and bad leads.

# Approach 2: Transitive Deduplication Using Union-Find (Disjoint Set)

**How Transitive Deduplication Works:**

For Example,

```
Lead a = {"_id": "1", "email": "x@example.com", "firstName": "A", "lastName": "X", "address": "Addr1", "entryDate": "2024-01-01T00:00:00Z"}
Lead b = {"_id": "2", "email": "x@example.com", "firstName": "B", "lastName": "Y", "address": "Addr2", "entryDate": "2024-01-02T00:00:00Z"}
Lead c = {"_id": "1", "email": "z@example.com", "firstName": "C", "lastName": "Z", "address": "Addr3", "entryDate": "2024-01-03T00:00:00Z"}
```

Let's Understand:

- a and b share email, so they are grouped.
- a and c share _id, so c joins the same group.
- Hence, a, b, c becomes 1 transitive group. (All the 3 Leads are Duplicates)
- Since Lead c, has the latest entryDate, it is selected as the preferred lead.

Approach 2 uses a **Union-Find (Disjoint Set)** data structure to detect clusters of duplicates connected via shared _id or email. It groups leads into sets where any two leads are linked transitively by common _id or email.

Steps:

- Separate out bad leads (missing _id or email).
- Assign each valid lead an index.
- For each lead:
  - Union leads sharing the same _id.
  - Union leads sharing the same email.
- After unions, leads belonging to the same root form a group.
- For each group:
  - Select the preferred lead (e.g., latest entryDate).
  - Log changes between preferred and other leads in the group.
  - Add preferred lead to final output maps.

This approach resolves **transitive duplicates** comprehensively.

**Algorithm:**

1. Filter out bad leads.
2. Initialize Union-Find data structure for all valid leads.
3. Map _id and email to lead indices, union leads with same _id or email.
4. Identify groups by finding roots in Union-Find.
5. For each group:
   - Determine preferred lead.
   - Log changes.
   - Add preferred lead to output maps.
6. Return deduplicated leads and bad leads.

## Pros and Cons

| Aspect | Approach 1: Simple Map-Based Deduplication | Approach 2: Union-Find (Transitive) Deduplication |
|---|---|---|
| Pros | Simple and straightforward to implement. | Detects all duplicates including transitive links. |
| | Efficient for datasets with direct _id or email duplicates. | More complete and accurate deduplication. |
| | Easy to maintain with minimal data structures. | Handles complex real-world data with overlapping identifiers. |
| Cons | cannot detect or merge transitive duplicates. | More complex to implement. |
| | May leave duplicates unresolved if identifiers are indirectly shared. | Slightly higher computational overhead due to union-find operations. |
| | Less robust for complex real-world data. | Requires extra memory to track union-find structures. |

## Algorithmic Complexity

| Aspect | Approach 1 | Approach 2 (Union-Find) |
|---|---|---|
| **Time Complexity** | `O(n)` for scanning + `O(1)` lookups per lead = **O(n)** | Union-Find with path compression = **O(n * α(n))** (very close to linear) |
| **Space Complexity** | `O(n)` for storing maps and leads | `O(n)` for storing parent maps and lead clusters |

Where n is the number of leads, and α(n) is the inverse Ackermann function, practically ≤ 5.

## Conclusion: Why Approach 2 Was Selected

Approach 2 was selected for this project because:

- It **handles transitive duplicates** which are common in real-world datasets where duplicates share identifiers indirectly.
- Provides a **more comprehensive and reliable** deduplication.
- Ensures **no duplicates are left unresolved**, preventing data quality issues downstream.
- Though more complex, its additional robustness outweighs the minor performance costs.

## Understanding Algorithms with Sample Input

Below is a sample input dataset illustrating lead records with overlapping _id and emails:

```
{
  "leads":[
    {"_id": "jkj238238jdsnfsj23", "email": "foo@bar.com", "firstName": "John", "lastName":
"Smith", "address": "123 Street St", "entryDate": "2014-05-07T17:30:20+00:00"},
    {"_id": "edu45238jdsnfsj23", "email": "mae@bar.com", "firstName": "Ted", "lastName":
"Masters", "address": "44 North Hampton St", "entryDate": "2014-05-07T17:31:20+00:00"},
    {"_id": "wabaj238238jdsnfsj23", "email": "bog@bar.com", "firstName": "Fran", "lastName":
"Jones", "address": "8803 Dark St", "entryDate": "2014-05-07T17:31:20+00:00"},
    {"_id": "jkj238238jdsnfsj23", "email": "coo@bar.com", "firstName": "Ted", "lastName":
"Jones", "address": "456 Neat St", "entryDate": "2014-05-07T17:32:20+00:00"},
    {"_id": "sel045238jdsnfsj23", "email": "foo@bar.com", "firstName": "John", "lastName":
"Smith", "address": "123 Street St", "entryDate": "2014-05-07T17:32:20+00:00"},
    {"_id": "qest38238jdsnfsj23", "email": "foo@bar.com", "firstName": "John", "lastName":
"Smith", "address": "123 Street St", "entryDate": "2014-05-07T17:32:20+00:00"},
    {"_id": "vug789238jdsnfsj23", "email": "foo1@bar.com", "firstName": "Blake", "lastName":
"Douglas", "address": "123 Reach St", "entryDate": "2014-05-07T17:33:20+00:00"},
    {"_id": "wuj08238jdsnfsj23", "email": "foo@bar.com", "firstName": "Micah", "lastName":
"Valmer", "address": "123 Street St", "entryDate": "2014-05-07T17:33:20+00:00"},
    {"_id": "belr28238jdsnfsj23", "email": "mae@bar.com", "firstName": "Tallulah", "lastName":
"Smith", "address": "123 Water St", "entryDate": "2014-05-07T17:33:20+00:00"},
    {"_id": "jkj238238jdsnfsj23", "email": "foo@bar.com", "firstName": "John", "lastName":
"Smith", "address": "123 Street St", "entryDate": "2014-05-07T17:33:20+00:00"}
  ]
}
```

### How Approach 1 Processes This Input

**Step-by-Step:**

1. Insert lead 1 → new _id, added.
2. Insert lead 2 → new _id, **but email matches** existing: maps email to existing _id=1, dedupes with lead 1.
3. Insert lead 1 again → _id already exists, dedupes.
4. Insert lead 3 → new _id, added.
5. Insert lead 4 → new _id, added.

**Issues:**

- Deduplication is local/pairwise.
- Email-based deduplication works only once (emailToIdMap can only point to one _id).
- Transitive chain 1 ↔ 2 ↔ 3 is **not fully resolved**.
- Retains 1, 3, 4 as separate leads.

### How Approach 2 Processes This Input

**Step-by-Step (Union-Find):**

1. Insert 1 → own set
2. Insert 2 → union with 1 (via email)
3. Insert 1 again → already in group
4. Insert 3 → own group
5. Insert 4 → own group

**Final Clusters:**

- **Group A**: {1, 2} via email → plus 1 and c → full group = {a, b, c}
- **Group B**: {3}
- **Group C**: {4}

**Select latest per group:**

- Group A → Lead c (latest)
- Group B → Lead 3
- Group C → Lead 4

**Summary Table:**

| Feature | Approach 1 (Simple Map) | Approach 2 (Union-Find) |
|---|---|---|
| Handles transitive duplicates | No | Yes |
| Time complexity | `O(n)` | `O(n * α(n))` |
| Space complexity | `O(n)` | `O(n)` |
| Final entries in sample input | 5 | 4 |
| Accuracy of deduplication | Partial (pairwise only) | Full (transitive merge) |