Efficient Online User Tracking

Martin Elsman
Zecure.com

July 2006

Abstract

We present a technique for identifying users efficiently and tracking user behaviour. The technique makes use of a union-find data structure for recording incoming events that provide evidence that other events in fact come from the same user. The data structure is implemented in a relational database. We describe the operations on this data structure and show how the events that are inferred to come from a particular user can be retrieved efficiently.

1 Introduction

Users of Internet Web sites are identified differently over time. For example, when a user first visits a Web site, the user may only be identified by the IP address at which the user is located (possibly a proxy) and perhaps by browser type and screen resolution (if Javascript is supported by the browser). At the first response, a user's browser may have received a session cookie, which enables a Web application to conclude that future requests made by the user in the session are indeed made by that user. At some point during the session, the user may sign up for a service by entering an email address, or login to the system using a login name and a password for the system. In both cases, the user reveals information that can be associated with the session cookie. Over time, different session cookies are associated with the user, but all information associated with the session cookies may be relevant for user tracking and user behavior analysis.

1.1 Events and Elements

An *event type* has a name and is associated with zero or more *element types*, each of which consists of a name, an associated regular expression, and an integer indicating whether a particular element uniquely identifies a user. A value of zero indicates that the element does not uniquely identify a user. Larger integers guides which element value is used for printing (see the discussion below).

For recording requests (made by a user's browser) and other facts, we use the concept of an event. An *event* is associated with an event type ϵ, a particular

point in time, and with zero or more elements, each of which is associated with an element type and a value (which must match the regular expression of the associated element type). The element types associated with an event type ϵ dictates the elements that must be associated with an event with event type ϵ.

Whereas event types and element types are defined once for a particular Web site, events and event elements are dynamic entities, which are created when a user browses the Web site. For example, we may define the following element types:

Element type	**Regular expression**	**Unique**	**Description**
Cookie	`[a-zA-Z0-9]+`	1	Cookie stored in user's browser.
Email	`[.@a-zA-Z0-9_-]+`	2	User email address.
URL	`[.a-z:/A-Z0-9_%-]+`	0	URL the user is redirected from.

Based on the above element types, we may define the following event types:

Event type	**Element Types**	**Description**
Redirect	[Cookie, URL]	User is redirected from a foreign web site.
Browse product	[Cookie]	User browses a product.
Add to cart	[Cookie]	User adds a product to cart.
Checkout	[Cookie, Email]	User orders content of cart—receipt sent to user.
Bounce	[Email]	Email bounces.

Given the above event types, consider the following sequence of events:

Time	**Event Type**	**Elements** (on the form value:type)
t	Redirect	[`aa`:Cookie, `www.google.com`:URL]
t+2s	Redirect	[`ab`:Cookie]
t+3s	Add to cart	[`ab`:Cookie]
t+4s	Browse product	[`aa`:Cookie]
t+4s	Checkout	[`ab`:Cookie, `asdsa@adfs.com`:Email]
t+30s	Add to cart	[`aa`:Cookie]
t+32s	Bounce	[`asdsa@adfs.com`:Email]
t+52s	Checkout	[`aa`:Cookie, `hans@gmail.com`:Email]

The first event is registered at time t and the last event at time $t + 52s$. By simple reasoning, based on the fact that elements with element types Email and Cookie uniquely identifies a user, one can conclude that the user with email address `hans@gmail.com` is first redirected from `www.google.com`, where after the user browses a product, adds a product to a cart, and, finally, checks out the cart, with the entire session taking 52 seconds.

As we shall see in the following, we make use of a union-find data structure for establishing efficiently that two events comes from the same user.

We shall not be concerned here about how the events and associated elements are obtained (be it from a log file or by XML requests to the server hosting the technology), but rather focus on how events are represented and how user information is gathered and maintained, incrementally, so that all events related to a particular user may be found efficiently (in time almost proportinal to the number of events found). Moreover, we shall describe how new event information is added to the representation in almost constant time.

2 Outline

In Section 3, we present a data model for events, based on an underlying union-find data structure. In Section 4, we describe how a system can refer to a user independently of the id of the ECR. In Section 5, we describe how the events associated with a user may be retrieved efficiently. In Sections 6 and 7, we describe related and future work and conclude.

3 The Data Model

3.1 Element Types and Event Types

The following entity definitions define element types and event types. Event types are associated with element types through the entity `event_type_element_type`.

```
CREATE TABLE element_type (
   id           SERIAL PRIMARY KEY,
   name         VARCHAR UNIQUE NOT NULL,
   regex        VARCHAR NOT NULL,
   unique_ident INTEGER DEFAULT 0 NOT NULL,
);

CREATE TABLE event_type (
   id           SERIAL PRIMARY KEY,
   name         VARCHAR UNIQUE NOT NULL
);

CREATE TABLE event_type_element_type (
   id               SERIAL PRIMARY KEY,
   event_type_id    INTEGER REFERENCES event_type(id) NOT NULL,
   element_type_id  INTEGER REFERENCES element_type(id) NOT NULL,
   UNIQUE           ( event_type_id, element_type_id )
);
```

3.2 Union-Find Data Structure for Representing Users

The following entities constitute the heart of the union-find data structure. As we shall see later, each event is associated with an *equivalent class representative* (ECR), which is used for accumulating event information when events are added

incrementally. ECR's can be unioned efficiently, which means that events can be unioned efficiently (e.g., two events should be unioned when the events have identical session cookies or email addresses as elements).

We use Tarjan's union-find algorithm [4], which means that union and find operations work by updating and compressing links (using path-halving). Another possibility, which we shall return to in Section 7, would be to use explicit substitutions.

The data model for the union-find data structure consists of two entities `ecr_info` and `ecr_link`:

```
CREATE TABLE ecr_info (
   id                 BIGSERIAL PRIMARY KEY,
   ecr_rank           INTEGER NOT NULL DEFAULT 0,
   first_at           TIMESTAMP DEFAULT now() NOT NULL,
   last_at            TIMESTAMP DEFAULT now() NOT NULL,
   event_count        INTEGER NOT NULL DEFAULT 1,
   unique_ident_value VARCHAR, -- Name of user derived from
                               -- element with element_type of
                               -- highest unique_ident value.
   unique_ident_element_type_id INTEGER REFERENCES element_type(id),
   updated_at         TIMESTAMP DEFAULT now() NOT NULL
);

CREATE TABLE ecr_link (
   id                 BIGINT NOT NULL UNIQUE,
   next_id            BIGINT NOT NULL,
   UNIQUE             ( id, next_id )
);
```

Accumulated event information is stored in `ecr_info` rows. For instance, we keep track of the total number of events (`event_count`), the points in time for the first and last events (`first_at` and `last_at`), the element type with the highest `unique_ident` values, and the associated value for this element type (the last two are used for pretty printing a user).

We shall refer to rows in the `ecr_info` table as *ECR-nodes* and rows in the `ecr_link` table as *ECR-links*. Moreover, we shall refer to id's in the `ecr_info` and `ecr_link` tables as *ECR-ids*. An ECR-id is *valid* if it appears as id in a row in the `ecr_info` table.

The following function implements the find operation for the union-find data structure. The function takes as argument an ECR-id and returns a valid ECR-id in the sense that it refers to a row in the `ecr_info` table. The function implements path-compression, by *path halving*, a technique that allows for the recursive call to be a tail-call.

```
CREATE OR REPLACE FUNCTION
ecr_find ( aid bigint ) RETURNS bigint AS $$
  DECLARE
    nid bigint;
```

```
    nnid bigint;
  BEGIN
    SELECT INTO nid next_id FROM ecr_link WHERE id = aid;
    IF NOT FOUND THEN RETURN aid; END IF;

    -- path halving
    SELECT INTO nnid next_id FROM ecr_link WHERE id = nid;
    IF NOT FOUND THEN RETURN nid;
    ELSE
      UPDATE ecr_link SET next_id = nnid WHERE id = aid;
      RETURN ecr_find(nnid);    -- tail call!
    END IF;
  END;
$$ LANGUAGE plpgsql;
```

The data model allows for two ECR-nodes to be unioned as follows:

1. Choose the ECR-node with the lowest *rank* as the *surviving* ECR-node.
2. Merge information in the non-surviving ECR-node into the surviving ECR-node and increase the `ecr_rank` value in the surviving ECR-node by one.
3. Add an ECR-link from the non-surviving ECR-node to the surviving ECR-node.
4. Delete the non-surviving ECR-node.

Here is the function `ecr_info_union` for merging the information in the non-surviving ECR-node into the surviving ECR-node:

```
CREATE OR REPLACE FUNCTION
ecr_info_union( ecr_id1 bigint, ecr_id2 bigint, ecr_id_new bigint )
RETURNS void AS $$
  DECLARE
    e1 RECORD;
    e2 RECORD;
    first_at_new timestamp;
    last_at_new timestamp;
    event_count_new integer;
    unique_ident_new integer;
    unique_ident_value_new varchar;
    unique_ident_element_type_id_new integer;
    ecr_id_del bigint;
  BEGIN
    SELECT INTO e1 * FROM ecr_info WHERE id = ecr_id1;
    SELECT INTO e2 * FROM ecr_info WHERE id = ecr_id2;

    first_at_new := min_timestamp(e1.first_at,e2.first_at);
    last_at_new := max_timestamp(e1.last_at,e2.last_at);
    event_count_new := e1.event_count + e2.event_count;
    unique_ident_new := max_int(e1.unique_ident, e2.unique_ident);
```

```
    -- Find the best unique_ident_value
    IF unique_ident_new = e1.unique_ident THEN
      unique_ident_value_new := e1.unique_ident_value;
      unique_ident_element_type_id_new := e1.unique_ident_element_type_id;
    ELSE
      unique_ident_value_new := e2.unique_ident_value;
      unique_ident_element_type_id_new := e2.unique_ident_element_type_id;
    END IF;

    -- Update the ECR-node that was chosen as the new ECR
    UPDATE ecr_info
       SET first_at = first_at_new,
           last_at = last_at_new,
           event_count = event_count_new,
           unique_ident = unique_ident_new,
           unique_ident_value = unique_ident_value_new,
           unique_ident_element_type_id = unique_ident_element_type_id_new,
           ecr_rank = ecr_rank + 1,
           updated_at = now()
     WHERE id = ecr_id_new;

    -- Delete the no-longer used ECR-node
    IF ecr_id_new = ecr_id1 THEN ecr_id_del = ecr_id2;
    ELSE ecr_id_del = ecr_id1;
    END IF;
    DELETE FROM ecr_info WHERE id = ecr_id_del;
  END;
$$ LANGUAGE plpgsql;
```

When updating the surviving ECR-node, the `updated_at` field is also updated to the present time. The `update_at` value is used by the function in Section 5 for retrieving events made by a user.

The function `ecr_union`, for unifying two ECR-nodes (i.e., users), looks as follows:

```
CREATE OR REPLACE FUNCTION
ecr_union( id1 bigint, id2 bigint ) RETURNS bigint AS $$
  DECLARE
    ecr_id1 bigint;
    ecr_id2 bigint;
    ecr_id bigint;      -- new chosen ecr_id (either ecr_id1 or ecr_id2)
    ecr_id_link bigint; -- the ecr_id that was not chosen
    rank1 integer;
    rank2 integer;
  BEGIN
    ecr_id1 := ecr_find( id1 );
    ecr_id2 := ecr_find( id2 );
    IF ecr_id1 = ecr_id2 THEN RETURN ecr_id1; END IF;
```

```
    SELECT INTO rank1 ecr_rank FROM ecr_info WHERE id = ecr_id1;
    SELECT INTO rank2 ecr_rank FROM ecr_info WHERE id = ecr_id2;

    IF rank1 < rank2 THEN
      ecr_id := ecr_id1;
      ecr_id_link := ecr_id2;
    ELSE
      ecr_id := ecr_id2;
      ecr_id_link := ecr_id1;
    END IF;

    -- Choose ecr_id as new ecr; thus, let ecr_id_link point at ecr_id
    INSERT INTO ecr_link(id, next_id) VALUES (ecr_id_link, ecr_id);
    PERFORM ecr_info_union(ecr_id1,ecr_id2,ecr_id);
    RETURN ecr_id;
  END;
$$ LANGUAGE plpgsql;
```

Notice that the ECR-node with the lowest rank is chosen as the surviving ECR-node.

3.3 Events and Elements

The data model for events and elements consists of the tables `event` and `element`, each of which references the tables `event_type` and `element_type`, respectively.

```
  CREATE TABLE event (
     id                BIGSERIAL PRIMARY KEY,
     ecr_id            BIGINT,  -- References either ecr_link(id) or ecr_info(id)
     ecr_id_updated_at TIMESTAMP DEFAULT now(),  -- For using ecr_id as index
     created_at        TIMESTAMP DEFAULT now() NOT NULL,
     event_type_id     INTEGER REFERENCES event_type( id ) NOT NULL
  );
  CREATE TABLE element (
     id              BIGSERIAL PRIMARY KEY,
     event_id        BIGINT REFERENCES event( id ) NOT NULL,
     element_type_id INTEGER REFERENCES element_type( id ) NOT NULL,
     value           VARCHAR( 255 ) NOT NULL,
     UNIQUE( event_id, element_type_id )
  );
```

As we shall see in Section 4, the attribute `ecr_id_updated_at` in the `event` table is used for retrieving the events associated with a user efficiently.

Adding an event with associated elements to the data structure involves inserting one row in the `event` table, inserting a row in the `element` table for each element, and calling the following function `element_unify` with the `id` attribute for each element:

```
  CREATE OR REPLACE FUNCTION
  element_unify( the_element_id bigint ) RETURNS void AS $$
```

```
DECLARE
  r record;
  ecr record;
  ecr_id bigint;
BEGIN
  SELECT INTO r eet.unique_ident, ee.event_id,
                ee.element_type_id, ee.value, e.ecr_id
    FROM element_type eet
    LEFT JOIN element ee ON ee.element_type_id = eet.id
    LEFT JOIN event e ON e.id = ee.event_id
    WHERE ee.id = the_element_id;

  IF r.unique_ident > 0 THEN

    -- maybe add element value to ECR-node
    SELECT INTO ecr * FROM ecr_info WHERE id = ecr_find(r.ecr_id);
    IF r.unique_ident > ecr.unique_ident THEN
      UPDATE ecr_info
         SET unique_ident = r.unique_ident,
             unique_ident_value = r.value,
             unique_ident_element_type_id = r.element_type_id
       WHERE id = ecr.id;
    END IF;

    -- Look for another element with the same element type
    -- and the same element value. It is sufficient to look
    -- for just one such element (the argument for this claim
    -- uses simple induction).

    SELECT INTO ecr_id e.ecr_id
      FROM element ee2
      LEFT JOIN event e ON e.id = ee2.event_id
      WHERE ee2.value = r.value
      AND ee2.element_type_id = r.element_type_id
      AND (ee2.event_id != r.event_id)
      LIMIT 1;

    IF FOUND THEN PERFORM ecr_union(r.ecr_id,ecr_id); END IF;
  END IF;
END;
$$ LANGUAGE plpgsql;
```

The function `element_unify` is guided by the `unique_ident` attribute in the `element_type`; only elements that uniquely identify a user may result in ECR-nodes (i.e., users) being unified.

4 What Identifies a User?

A user is represented in the system by an ECR-node. However, it would not be convenient to refer to a user using an ECR-id, since a valid ECR-id can become invalid over time—when new events are added—and since it could make sense periodically to garbage collect non-referred-to ECR-links (by updating each referenced ECR-id in the datamodel with the result of executing a find operation on the given ECR-id).

Instead, we can use element values with element types that uniquely identifies a user to locate a particular user. For example, given the element type named `Email`, which is specified to identify uniquely a user, and a particular email address, say `hans@yahoo.com`, the system can identify the ECR-id that represents this user, if the user exists in the system.

5 Retrieving Events Related to a User

To find all events related to a customer (identified by `ecr_id`), we cannot simply execute

```
SELECT * FROM event WHERE ecr_id = the_ecr_id                    (1)
```

since an ECR-id in the event table may identify a link to another ECR-node or link (after a union of two ECR-nodes). On the other hand, using the correct

```
SELECT * FROM event WHERE ecr_find(ecr_id) = the_ecr_id
```

would be too expensive, since no index is used. To deal with this inefficiency, we allow for the following SQL statement to be executed just before `(1)` is executed:

```
UPDATE event SET ecr_id = ecr_find(ecr_id),
                 ecr_id_updated_at = now()
 WHERE ecr_id_updated_at <= ecr_updated_at                       (2)
   AND ecr_updated_at >= created_at                              (3)
```

Here `ecr_updated_at` is the timestamp of the ECR-node in question. Condition `(2)` specifies that we need only update ECR-ids pointing (transitively) to ECR-nodes that have been updated since the ECR-id was updated. Condition `(3)` specifies that we need only update ECR-ids for those events that were created before the pointed-to ECR-node was updated (due to a union).

Notice that b-tree indexes are used for making the above `UPDATE` statement efficient. Here is a function `ecr_id_update` for executing the above update statement based on an `ecr_updated_at` time stamp:

```
CREATE OR REPLACE FUNCTION
ecr_id_update ( ecr_updated_at timestamp ) RETURNS void AS $$
  BEGIN
     UPDATE event SET ecr_id = ecr_find(ecr_id),
                      ecr_id_updated_at = now()
```

```
      WHERE ecr_id_updated_at <= ecr_updated_at
        AND ecr_updated_at >= created_at;
  END;
$$ LANGUAGE plpgsql;
```

6 Related Work

There is much related work in the area of tracking user behavior in a Web-based environment and many commercial products available for user tracking, including Opentracker (`www.opentracker.net`), a cookie-based solution for monitoring user-behavior on a Web site. An alternative is the open source (Apache License) solution phpOpenTracker, which is a framework for the analysis of website traffic and visitor analysis, primarily focused on usability analysis.

As far as we are aware, none of the mentioned solutions allow for unifying two separate users based posterior information that the two users are actually identical.

Another area of related work deals with how events are collected. Atterer et al. [1] uses a proxy-server to intercept a wide series of user events by adding Javascript to pages automatically. Simpler solutions include inspecting log files, both in common log file format (CLF) and extended common log file format (ECLF) [3, 2].

7 Conclusion and Future Work

We have presented a technique for collecting user events online, which provide efficient access to events made by users and which supports the possibility for unifying two different users if posterior information indicates that the two users are actually identical.

There are plenty of possibilities for future work. For instance, the solution that we present is based on a union-find structure for implementing efficient substitution of user-ids when two users are to be considered identical. An alternative is to use explicit substitution, which, may ease the maintenance of an index on the `ecr_id` attribute of the `event` entity. However, such a solution would lead to an $O(N)$ union operation of two users, where N is the maximum number of events in an equivalence class.

Acknoledgments

Thanks to Bo Lorentsen for many great discussions concerning this work. Bo is in fact the architect behind the data models for element types, event types, elements, and events.

References

[1] R. Atterer, M. Wnuk, and A. Schmidt. Knowing the user's every move: user activity tracking for website usability evaluation and implicit interaction. In *15th international Conference on World Wide Web (WWW'06)*, pages 203–212. ACM Press, May 2006. Edinburgh, Scotland.

[2] Dan R. Greening. Tracking users—what marketers really want to know, July 1999. http://www.webtechniques.com/archives/1999/07/greening/.

[3] Philip Greenspun. *Philip and Alex's guide to Web publishing*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[4] Robert Endre Tarjan. *Data structures and network algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1983.

A Various `plpgsql` Utility Functions

```
CREATE OR REPLACE FUNCTION
min_timestamp(t1 timestamp,t2 timestamp) RETURNS timestamp AS $$
  BEGIN
    IF t1 < t2 THEN RETURN t1;
    ELSE RETURN t2;
    END IF;
  END;
$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION
max_timestamp(t1 timestamp,t2 timestamp) RETURNS timestamp AS $$
  BEGIN
    IF t1 > t2 THEN RETURN t1;
    ELSE RETURN t2;
    END IF;
  END;
$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION
max_int(t1 integer,t2 integer) RETURNS integer AS $$
  BEGIN
    IF t1 > t2 THEN RETURN t1;
    ELSE RETURN t2;
    END IF;
  END;
$$ LANGUAGE plpgsql;
```