DRAFT: Building `psguid` Map.

September 19, 2011

Abstract

User activity tracking is at the core of our business. Owing to Web Event Collection framework (WEC), our system receives detailed information about user online activity, e.g., clicking, searching, placing order, etc. While some activities such as placing an order require that a user be identified or authenticated, other activities such as clicking or searching do not require identification. Thus, once a user is identified we need to merge current activity with prior activity, i.e., activity which precedes identification. Prior activity is of great value. Intuitively, such activity is indicative of a user's proclivity.

Our current approach assigns to each user a Globally Unique Identifier (GUID) which we shall call `psguid`. `psguid`s are stored in third-party cookies inside a user agent, i.e., browser. WEC events tag each user activity with a `psguid`[1]. In addition, WEC events may contain user identifiying information, e.g., email address, user id. In order to consolidate all WEC events attributed to the same user, we compute sets of all related `psguid`s. Intuitively, `psguid`s are related if they are tracking the same user. Subsequently, we build the map `psguid` $\rightarrow$ `tid` which maps each `psguid` to its corresponding consumer `tid`[2]. The invariant is that all related `psguid`s must map to the same `tid`. Downstream processes use the map for event attribution; i.e., if two events contain related `psguid`s, then both events belong to the same user in virtue of having the same `tid`.

1 Introduction

Our internal jobs such as the *scoring engine* use `tid`s to identify each consumer. (Using database terminology, `psguid`s can be seen as surrogate keys, whereas `tid`s are natural keys.) Consequently, before user activity can be scored we must associate it with a unique `tid`. This association is captured by the `psguid` map. The construction of `psguid` map is accomplished by a `MapReduce` job implemented in `PSguidMapBuilder.java`. The main workflow consists of the following stages: *preprocessing, computing connected components, postprocessing.*

[1] Currently, three different fields in WEC may contain `psguid`s: `psrw`, `psrj`, `psguid`.

[2] `tid` is short for *table identifier*. Consumer `tid`s are essentially user identifiers. Internally, emails and user ids are mapped into `tid`s.

In preprocessing, we are given WEC records—these come in form of sequence files whose values are tab-delimited (keys are ignored). If there exists `psguid` map, e.g., from previous run, then it serves as input to the preprocessing stage. Essentially, preprocessing determines which `psguid`s and `tid`s are *immediately* related. This relation is distributed amongst the sets of nodes which form the output of the preprocessing stage. (See Sect. 2 for details.)

Next, we execute a `MapReduce` job which computes all *transitively* related `psguid`s and `tid`s; i.e., if x, y are related and y, z are related, then x, z are transitively related. All related `psguid`s and `tid`s form a connected component in the undirected graph whose edges are (unordered) pairs of `psguid`s which are immediately related, i.e., related due to some WEC record.

During the final stage, namely postprocessing, we map each `psguid` to a unique `tid`. (See Sect. 5 for details.) That is, the postprocessing job outputs **key**=`psguid`, **value**=`tid` records; the output is then used to upload (and overwrite) the `psguid` map.

2 Preprocessing

There are two different preprocessors: WEC and Map. The latter is executed only if there is an existing `psguid` map. In case of WEC, the basic idea is to extract values of fields from WEC input which are related according to the following relation[3]: x, y are related iff there exists a WEC record r such that

$$(r.\texttt{psrw} = x \lor r.\texttt{psrj} = x \lor r.\texttt{psguid} = x \lor r.\texttt{tid}_1 = x \lor r.\texttt{tid}_2 = x) \land$$
$$(r.\texttt{psrw} = y \lor r.\texttt{psrj} = y \lor r.\texttt{psguid} = y \lor r.\texttt{tid}_1 = y \lor r.\texttt{tid}_2 = y)$$

where $\texttt{tid}_1$, $\texttt{tid}_2$ are technically not in r but rather resolved from email and user maps, respectively. For example, if a WEC record contains two distinct `psguid`s, say x, y and the record resolves to a single `tid`, say z, then x, y, z are related; the output of WEC preprocessing in this example is the set $\{x, y, z\}$.

In case of Map preprocessing, x, y are related iff both x, y are `psguid`s and map to the same `tid` or x (resp., y) is a `psguid` and y (resp., x) is the corresponding `tid`. For example, suppose we have $(\texttt{psguid}_1, \texttt{tid})$, $(\texttt{psguid}_2, \texttt{tid})$ in the map. Then, $\texttt{psguid}_1$, $\texttt{psguid}_2$, `tid` are (pairwise) related.

2.1 WEC

Preprocessing of the WEC input is handled by a `MapReduce` job implemented in `WECPreprocessor.java`. The input to the job is a set of sequence files whose values consist of tab-delimited fields[4]. In addition, *email* and *user* maps[5] are specified; they map an email (resp., user) string to a `tid`. The output of the job is a sequence file whose values are sets of nodes, where each node is either a `guid` or a `tid`. The job has zero reducers and an arbitrary number of mappers.

[3]Note, the relation is symmetric and reflexive, but may not be transitive.

[4]The mapping from field names to positions can be found in `wec.xml`.

[5]These maps must be configured on a per client basis.

In WEC preprocessing, each row of WEC input is examined in order to extract related fields. Typically, these fields are `psrw`, `psrj`, and `psguid`. The fields denote related `guids` which may be pairwise distinct. In addition, the fields `email`, `consumer_guid`, `current_url`, `guest_order` are examined for the purpose of extracting the identity of the user. If either `email` or `current_url` contains a nonempty string, then we consult the given email map to lookup a `tid` corresponding to the email. (Only if `email` yields an empty string or the corresponding lookup fails will the `current_url` be examined.) Furthermore, if `consumer_guid` yields a nonempty value, then the given user map is consulted to lookup the corresponding `tid`. (Note, if `guest_order` contains the character `y`, ignoring case, the email map is used instead of the user map.) Consequently, the resolved `tids` are output in conjunction with the `guids`, i.e., `psrw`, `psrj`, `psguid`. It is indeed possible[6] to obtain two distinct `tids`, one corresponding to email, the other corresponding to user id. In case of distinct `tids`, both are output, and an appropriate counter[7] is incremented.

3 Map

Preprocessing of the existing `psguid` $\rightarrow$ `tid` is handled by a `MapReduce` job implemented in `PSguidMapPreprocessor.java`. The input to the job is a set of comma-delimited values. Each value is essentially composed of (`psguid`, `tid`) pairs. The output of the job is of the same format as the output of `WECPreprocessor`, namely it is a sequence file whose values comprise sets of nodes. The job has an arbitrary number of mappers and a given[8] number of reducers.

Each mapper simply parses the comma-delimited values and reverses each pair, i.e., **key**=`tid`, **value**=`psguid`. Each reducer unions all `psguids` for a given `tid` and outputs the corresponding set of nodes, i.e., $\{\texttt{guid}_1, \ldots, \texttt{guid}_n, \texttt{tid}\}$. If any data inconsistencies are encountered, e.g., `tids` should be *positive* `long` values, appropriate counters are incremented.

4 Connected Components

After preprocessing we have a bag of sets of related `guids` and their corresponding `tids`. More abstractly, each set denotes a set of connected nodes in an undirected graph. The connected components are computed by a `MapReduce` job implemented in `ElectionPartition.java`. The job is iterative and proceeds in alternating stages, namely *election* and *partition*. Roughly speaking, the election job finds all intersecting sets and replaces them by their union; the partition job determines pairwise disjoint sets. The process iterates until all disjoint sets have been computed. Initially, each set in the input is marked

[6] This case seems to be rare but should be investigated further.

[7] Several data validation counters are maintained; see the source code for details.

[8] Number of reducers, once configured remains constant for all the intermediate jobs; the default is 5.

`nondisjoint`. As each partition job determines that a set is disjoint it is marked `disjoint`. Thus, the connected components are contained amongst the output files of the partition jobs. For details, see Sect. 7 and the source code.

5 Postprocessing

Postprocessing is handled by a `MapReduce` job implemented in `Postprocessor.java`. The input to the job is a set of pairwise disjoint sets of nodes. The output is the new `psguid` $\rightarrow$ `tid` map. Thus, each output key, value pair is of the form **key**=`psguid`, **value**=`tid`.

Recall, we began with sets of related `guid`s and their corresponding `tid`s. At this ime, we have computed *all* related `psguid`s which from connected components. What remains is to output each `psguid` along with the `tid` it is mapped to. Thus, for each disjoint set S, and for each `psguid`, x, such that $x \in S$, we output **key**=x, **value**=`tid` where `tid` is obtained as follows. If S contains multiple `tid`s, we attempt to grab any `tid` which came[9] from WEC. If no such `tid` exists, then an arbitrary `tid` (in S) is returned. In case S contains no tids, we generate a fresh one by calling the *sequence* server. (Sequence server relies on `Oracle`'s sequences to atomically generate the next sequence number.)

Note, we must generate `tid`s because jobs such as the *scoring engine* are reliant on `tid`s. The fact that `tid`s from WEC have precedence over those from MAP is favorable because typically WEC follows MAP (in time); e.g., MAP may contain anonymous `tid`s obtained from the sequence server which, owing to precedence, would be replaced by `tid`s which resolve to email or user id.

6 Discussion

Open problems.

- Multiple users using the same user agent residing on the same physical device are conflated, i.e., treated as a single user.
- Same `psguid` is shared across distinct user agents, e.g., `Firefox` and `MSIE` which results in the case of conflated identity, as in the above. In theory, this case should be extremely rare if it is due to `psguid` collisions. However, this cases occurs with a rather high frequency in our current system. Upon investigation, it was hypothesized that some sort of proxy caching is taking place. Further investigation is needed.
- A given WEC record may resolve to two distinct `tid`s. This seems to be rare but should be investigated.

[9] Each `tid` is tagged with its source, i.e., WEC or MAP.

```
for each n ∈ N
  MAKE−SET(n)
for each (u, v) ∈ E
  if FIND−SET(u) ≠ FIND−SET(v)
    UNION(u, v)
```

Figure 7.1: Computing connected components using union-find data structure.

7 Appendix

Connected Components. Suppose we are given an undirected graph $G = (N, E)$, where N is a set of nodes, E is an unordered set of node pairs. Below we define connected components of G.

Definition 7.1 (connected components) Connected components of G are defined inductively:

- if $\{u, v\} \in E$, then u, v are in the same component C
- if $u \in C$, and there exists v such that $\{u, v\} \in E$, then $v \in C$

The following lemma captures the fact that all connected components are pairwise disjoint.

Lemma 7.1 *For any two connected components C_i, C_j, if $C_i \cap C_j \neq \emptyset$, then $C_i = C_j$.*

Example 7.1 Suppose the graph consists of the following edges: $\{1, 2\}$, $\{2, 3\}$, $\{3, 4\}$, $\{5\}$. Then, the connected components are the following disjoint sets: $\{1, 2, 3, 4\}$, $\{5\}$.

Computing connected components. It is not difficult to show that connected components of undirected graphs can be computed using breadth-first search or depth-first search. (A single traversal suffices.)

Fig. 7.1 shows the pseudocode for another algorithm which uses the classical union-find data structure. The method MAKE−SET(n) creates the singleton set $\{n\}$; FIND−SET(u) returns a set representative for the disjoint set containing u; UNION(u, v) merges the sets containing u and v into a new disjoint set. Fig 7.2 shows the algorithm in action for the graph used in Example 7.1.

We can also compute connected components without the use of union-find data structures. Suppose an edge relation is given indirectly in terms of any sets of connected nodes rather than pairs. E.g., all pairs of edges between the vertices u, v, w can be represented compactly as the set $\{u, v, w\}$. This representation inspires the following algorithm in Fig. 7.3.

The algorithm in Fig. 7.3 is correct owing to these observations. The initial sets of nodes are in fact included in the connected components. That is, for any initial set S, there exists some connected component C such that $S \subseteq C$.

Considered Edge	Resulting Disjoint Sets	
	$\{1\},\{2\},\{3\},\{4\},\{5\}$	(initial sets)
$\{1,2\}$	$\{1,2\},\{3\},\{4\},\{5\}$	
$\{2,3\}$	$\{1,2,3\},\{4\},\{5\}$	
$\{3,4\}$	$\{1,2,3,4\},\{5\}$	

Figure 7.2: Result of executing the algorithm in Fig. 7.1 on the graph in Example 7.1.

```
repeat

    choose S_i, S_j such that S_i ∩ S_j ≠ ∅:
        replace S_i, S_j by S_i ∪ S_j

until all S_i, S_j are pairwise disjoint
```

Figure 7.3: Computing connected components without union-find data structure.

Now, if a pair of sets intersects, i.e., $S_i \cap S_j \neq \emptyset$, then $S_i \subseteq C$ and $S_j \subseteq C$ for some connected component C. (The fact follows from Lemma 7.1.) Therefore, $S_i \cup S_j \subseteq C$; i.e., the union of intersecting sets must be included in the connected component. Since the sets are finite and each iteration computes a potentially larger subset of any given connected component, the loop must terminate precisely when all the sets are disjoint. The disjoint sets correspond to the connected components.

Number of iterations. Let n be the cardinality of the largest connected component. Then, the number of iterations in Fig. 7.3 can be bounded by $\mathcal{O}(\log n)$. Intuitively, each subsequent iteration computes sets whose cardinality is $\leq 2 * k_{i-1} - 1$ where k_i is the maximum cardinality of any set computed in iteration i. E.g., given the initial sets $\{a,b,c\}$, $\{c,d\}$, $\{a,e\}$, $\{f\}$, we have $k_0 = 3$. After the first iteration of the outer loop, the sets are: $\{a,b,c,d,e\}$, $\{f\}$ and hence $k_1 = 2 * k_0 - 1 = 5$.

Connected components in Hadoop. Fig. 7.4 has the pseudocode of the algorithm implemented in `ElectionPartition.java`. The algorithm is an adaptation[10] of the classical, (i.e., non-distributed) algorithm in Fig. 7.3, whence it has the same bound on the number of iterations, namely $\mathcal{O}(\log n)$ where n is the length of the longest path in the input graph.

[10]The distributed algorithm computes *all* pairs of intersecting sets in parallel, essentially replacing the non-deterministic choice in Fig. 7.3 with a constant number of iterations.

```
while(true) {
  run election job;
  if (optimization && numNondisjointSets <= THRESHOLD) {
    // compute connected components in memory using BFS
    run connected components job;
    break;
  }
  run partition job;
  if (numNondisjointSets == 0) {
    break;
  }
}
```

Figure 7.4: Pseudocode of Election-Partition algorithm.

Initially, all input sets are marked *nondisjoint*, and we begin the first iteration of election and partition jobs. The output of the election job becomes input for the partition job; the output of the partition job becomes input for the election job. Since new disjoint sets are identified in the partition job they must be filtered by subsequent iterations; the election job skips all disjoint sets.

Election Mapper.

- For each nondisjoint set S, elect a representative, $\mathsf{rep}(S)$. In the implementation we use the minimum[11] element in the set as the set's representative, i.e., $\mathsf{rep}(S) = \min(S)$. (Alternatively, we could use the maximum element without affecting the algorithm's correctness.)

- For each set S, output the following key,value pairs: $\mathsf{rep}(S) \to S$, $u_1 \to \{rep(S)\}$, $\ldots$, $u_n \to \{rep(S)\}$, for each $u_i \in S$ such that $u_i \neq \mathsf{rep}(S)$.

Note that each element of each set occurs at least once amongst all keys in the output. If a set is disjoint from all other sets, then each of its elements must occur exactly once amongst all keys in the output. Otherwise, if $S \cap T \neq \emptyset$, then at least the element u such that $u \in S \cap T$ must occur multiple times amongst all keys in the output; i.e., there are key,value pairs: $u \to \mathsf{rep}(S)$ and $u \to \mathsf{rep}(T)$.

Election Reducer. The reducer merely unions all sets which are mapped to by the same key.

Partition Mapper.

- For each key,value pair $u \to S$ such that $|S| = 1$, i.e., $S = \{v\}$ for some v, output $v \to \{u\}$.

[11] We require that an ordering on set elements is given.

- For each key,value pair $u \rightarrow S$ such that $|S| > 1$, output $u \rightarrow S$.

Partition Reducer. For each key and the corresponding sequence of sets, $S_1, \ldots, S_n$,

- let $S = S_1 \cup \ldots \cup S_n$
- if each set element occurs exactly twice in the sequence, then mark S *disjoint*; otherwise mark S *nondisjoint*
- output **value**=S (the key is null)

Example 7.2 Fig. 7.5 shows an example which is also used as a unit test in `TestElectionPartition.java`. (Sets are in square brackets; arrows denote key,value pairs; lack of arrows denote null keys.) The example illustrates the output of running election and partition jobs on the given sets using 4 iterations. Indeed, only 3 iterations are sufficient to compute all connected components. We used an extra iteration only for illustration—last iteration produces no output since the disjoint sets are skipped. Note that the singleton disjoint set $\{5\}$ is identified in the first partition job. The other disjoint set is identified in the third partition job. (No disjoint sets are identified in the second partition.) Thus, the output of this job is the union of outputs of all partition jobs.

Example 7.3 Fig. 7.5 shows an example wherein all input sets are already disjoint. In this case, one iteration suffices.

```
Starting Election/Partition with 4 iterations.
Using the following input:
[1, 2]
[2, 3]
[3, 4]
[5]

Results of Election-0
1-->[1, 2]
2-->[1, 2, 3]
3-->[2, 3, 4]
4-->[3]
5-->[5]

Results of Partition-0
[1, 2]
[1, 2, 3]
[2, 3, 4]
[5]

Results of Election-1
1-->[1, 2, 3]
2-->[1, 2, 3, 4]
3-->[1, 2]
4-->[2]

Results of Partition-1
[1, 2, 3]
[1, 2, 3, 4]
[1, 2, 3]

Results of Election-2
1-->[1, 2, 3, 4]
2-->[1]
3-->[1]
4-->[1]

Results of Partition-2
[1, 2, 3, 4]

Results of Election-3

Results of Partition-3
```

Figure 7.5: Example of running `ElectionPartition`.

```
Starting Election/Partition with 1 iterations.
Using the following input:
[1]
[2]
[3]

Results of Election-0
1-->[1]
2-->[2]
3-->[3]

Results of Partition-0
[1]
[2]
[3]
```

Figure 7.6: Example of running `ElectionPartition`.