A Knowledge Base for Personal Information Management

David Montoya Thomas Pellissier Tanon Serge Abiteboul
Pierre Senellart Fabian M. Suchanek

December 21, 2016

Internet users have personal data spread over several devices and across several online systems. In this paper, we introduce a novel framework for integrating a user's data from different sources into a single knowledge base. Our framework integrates data of different kinds into a coherent whole, starting with email messages, calendar, contacts, and location history. We show how event periods in the user's location data can be detected, how they can be aligned with events from the calendar, and how they can be linked to relevant emails. This allows users to query their personal information within and across different dimensions, and to perform analytics over their emails, events, and locations. Our system extends the `schema.org` vocabulary and provides a SPARQL interface.

1 Introduction

Internet users commonly have their personal data spread over several devices and services. This includes emails, messages, contact lists, calendars, location histories, and many other. However, commercial systems often function as *data traps*, where it is easy to check information in but difficult to query and exploit it. For example, a user may have all her emails stored with an email provider – but cannot find out which of her colleagues she interacts most frequently with. She may have all her location history on her phone – but cannot find out which of her friends' places she spends the most time at. Thus, a user often has paradoxically no means to make full use of data that she has created or provided. As more and more of our lives happen in the digital sphere, users are actually giving away part of their life to external data services.

We aim to put the user back in control of her own data. We introduce a novel framework that integrates and enriches personal information from different sources into a single knowledge base (KB) that lives on the user's machine, a machine she controls. Our system, Thymeflow, replicates data of different kinds from outside services and thus acts as a digital home for personal data. This provides the user with a high-level global view of that data, which she can use for querying and analysis. All of this integration and analysis happens locally on the user's computer, thus guaranteeing her privacy.

Designing such a personal KB is not easy: Data of completely different nature has to be modeled in a uniform manner, pulled into the knowledge base, and integrated with other data. For example, we have to find out that the same person appears with different email addresses in address books from different sources. Standard KB alignment algorithms do not perform well in our scenario, as we show in our experiments. Furthermore, integration spans data of different modalities: to create a coherent user experience, we need to align calendar events (temporal information) with GPS traces (location data) and place names.

We provide a fully functional and publicly available personal knowledge management system. A first contribution of our work is the management of location data. Such information is becoming commonly available through the use of mobile applications such as Google's Location History [13]. We believe that such data becomes useful only if it is semantically enriched with events and people in the user's personal space. We provide such an enrichment.

A second contribution is the adaptation of ontology alignment techniques to the context of personal KBs. The alignment of persons and organizations is rather standard. More novel are alignments based on time (a meeting in calendar and a GPS position), or space (an address in contacts and a GPS position).

Our third contribution is an architecture that allows the integration of heterogeneous personal data sources into a coherent whole. This includes the design of incremental synchronization, where a change in a data source triggers the loading and treatment of just these changes in the central KB. Inversely, the user is able to perform updates on the KB, which are persisted wherever possible in the sources. We also show how to integrate knowledge enrichment components into this process, such as entity resolution and spatio-temporal alignments.

As implemented, our system can provide answers to questions such as: Who have I contacted the most in the past month (requires alignments of different email addresses)? How many times did I go to Alice's place last year (requires alignment between contact list and location history)? Where did I have lunch with Alice last week (requires alignment between calendar and location history)?

Sec. 2 describes our data model and data sources and Sec. 3 the system architecture. Sec. 4 details our knowledge enrichment processes, while Sec. 5 discusses experimental results and Sec. 6 related work.

2 Data Model

In this section, we briefly describe the schema of the knowledge base, relying on RDF, and discuss the mapping of data sources to that schema.

Schema. We use the RDF standard [8] for knowledge representation. We use the namespace prefixes `wd` for `http://www.wikidata.org/entity/`, `schema` for `http://schema.org/`, and `rdf` and `rdfs` for the standard namespaces of RDF and RDF Schema, respec-

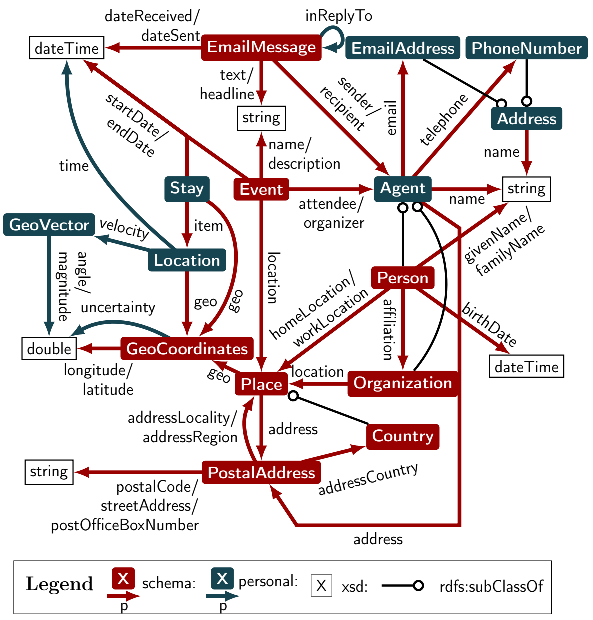

Figure 1: Personal Data Model

tively. A *named graph* is a set of RDF triples associated with a URI (its name). A *knowledge base* (KB) is a set of named graphs.

For modeling personal information, we use the `schema.org` vocabulary when possible. This vocabulary is supported by Google, Microsoft, Yahoo, and Yandex, and documented online. Wherever this vocabulary is not fine-grained enough for our purposes, we complement it with our own vocabulary, that lives in the namespace `http://thymeflow.com/personal#` with prefix `personal`.

Fig. 1 illustrates a part of our schema. Nodes represent classes, rounded colored ones are non-literal classes, and an edge with label p from X to Y means that the predicate p links instances of X to instances of type Y. We use locations, people, organizations, and events from `schema.org`, and complement them with more fine-grained types such as Stay, EmailAddress, and PhoneNumber. People and Organization classes are aggregated into a `personal:Agent` class.

Emails and contacts. We treat emails in the RFC 822 format [7]. An email is represented as a resource of type `schema:Email` with properties such as `schema:sender`, `personal:primaryRecipient`, and `personal:copyRecipient`, which link to `personal:Agent` instances. Other properties are included for the subject, the sent and received dates, the body, the attachments, the threads, etc.

Email addresses are great sources of knowledge. An email address such as "jane.doe@inria.fr" provides the given and family names of a person, as well as her affiliation. However, some email addresses provide less knowledge and some almost none, e.g., "j4569@gmail.com". Sometimes, email fields contain a name, as in "Jane

Doe <j4569@gmail.com>", which gives us a name triple. In our model, `personal:Agent` instances extracted from emails with the same combination of email address and name are considered indistinguishable (i.e., they are represented by the same URI). An email address does not necessarily belong to an individual; it can also belong to an organization, as in edbt-school-2013@imag.fr or fancy_pizza@gmail.com. This is why, for instance, the sender, in our data model, is a `personal:Agent`, and not a `schema:Person`.

A vCard contact [23] is represented as an instance of `personal:Agent` with properties such as `schema:familyName`, and `schema:address`. We normalize telephone numbers, based on a country setting provided by the user.

Calendar. The iCalendar format [9] can represent events. We model them as instances of `schema:Event`, with properties such as name, location, organizer, attendee, and date. The location is typically given as a postal address, and we will discuss later how to associate it to geo-coordinates and richer place semantics. The Facebook Graph API [10] also models events the user is attending or interested in, with richer location data and list of attendees (a list of names).

Locations. Smartphones are capable of tracking the user's location over time using different positioning technologies: satellite navigation, Wi-Fi, and cellular. Location history applications continuously run in the background, and store the user's location either locally or on a distant server. Each point in the user's location history is represented by time, longitude, latitude, and horizontal accuracy (the measurement's standard error). As no standardized protocol exists for managing a location history, we use the Google Location History format, in JSON, as Google users can easily export their history in this format. A point is represented by a resource of type `personal:Location` with properties `schema:geo`, for geographic coordinates with accuracy, and `personal:time` for time.

3 System Architecture

A personal knowledge base could be seen as a *view* defined over personal information sources. The user would query this view in a mediation style [11] and the data would be loaded only on demand. However, accessing, analyzing and integrating these data sources on the fly would be expensive tasks. For this reason, Thymeflow uses a warehousing approach. Data is loaded from external sources into a persistent store and then enriched.

Thymeflow is a Scala application that the user installs, providing it with a list of data sources, together with credentials to access them (such as tokens or passwords). The system accesses the data sources and pulls in the data. All code runs locally on the user's machine. None of the data leaves the user's computer. Thus, the user remains in complete control of her data. The system uses adapters to access the sources, and to transform the data into RDF. We store the data in a persistent Sesame-based triple store [3].

One of the main challenges in the creation of a personal KB is the temporal factor: data sources may change, and these updates should be reflected in the KB. Changes can happen during the initial load time, while the system is asleep, or after some inferences have

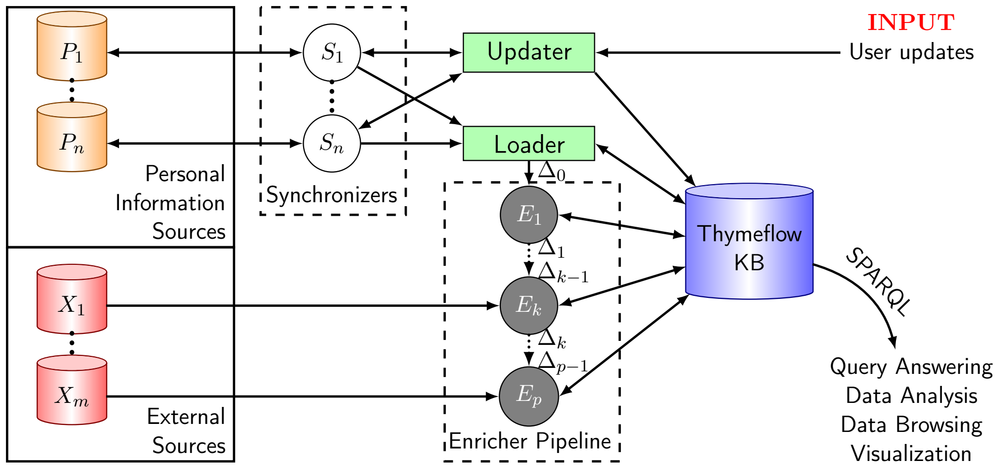

Figure 2: System architecture

already been computed. To address these dynamics, Thymeflow uses software modules called *synchronizers* and *enrichers*. Figure 2 shows synchronizers on the left, and enrichers in the center. Synchronizers are responsible for accessing data sources, enrichers (see Sec. 4) for inferring new statements, such as alignments between entities obtained by entity resolution.

Modules are scheduled dynamically and may be triggered by updates in the data sources (e.g., calendar entries) or by new pieces of information derived in the KB (e.g., the alignment of a position in the location history with a calendar event). The modules may also be started regularly for particularly costly alignment processes. When a synchronizer detects a change in a source, a pipeline of enricher modules is triggered, as shown in Figure 2. Enrichers can also use knowledge from external data sources, such as Wikidata [25], or OpenStreetMap.

Synchronizer modules are responsible for retrieving new data from a data source. For each data source that has been updated, the adapter for that particular source transforms the source updates since last synchronization into a set of insertions/deletions in RDF. This is of course relatively simple for data sources that provide an API supporting changes, e.g., CalDAV (calendar), CardDAV (contacts) and IMAP (email). For others, this requires more processing. The result of this process is a delta update, i.e., a set of updates to the KB since the last time that particular source was considered.

The KB records the provenance of each newly obtained piece of information. Synchronizers record a description of the data source, and enrichers record their own name. We use named graphs to store the provenance. For example, the statements extracted from an email message in the user's email server will be contained in a graph named with the concatenation of the server's email folder URL and the message id. The graph's URI is itself an instance of `personal:Document`, and is related to its source via the `personal:documentOf` property. The source is an instance of `personal:Source` and is in this case the email server's URL. Account information is included in an instance of `personal:Account` via the `personal:sourceOf` property. Account instances allows us to gather different kinds of data sources, (e.g., CardDAV, CalDAV and IMAP servers) belonging to one provider (e.g., corporate IT services) to which the user accesses through one identification. This provenance can be used to answer queries such as “What meetings were recorded in my work calendar for next Monday?”.

Finally, the system allows the propagation of information from the KB to the data sources. These can either be insertions/deletions derived by the enrichers, or insertions/deletions explicitly specified by the user. For instance, consider the information that different email addresses correspond to the same person. This information can be pushed to data sources, which may for example result in performing the merge of two contacts in the user’s list of contacts. To propagate the information to the source, we translate from the structure and terminology of the KB back to that of the data source and use the API of that source. The user has the means of controlling this propagation, e.g., specifying whether contact information in our system should be synchronized to her phone’s contact list.

The user can update directly the KB by inserting or deleting knowledge statements. Such updates to the KB are specified in the SPARQL Update language [12]. When no source is specified for recording this new information, the system considers all the sources that know the subject of the particular statement. For insertion, if no source is able to register a corresponding insertion, the system performs the insertion in a special locally persistent graph, called the *overwrite graph*. For deletions, if one source fails to perform a deletion (e.g., because the statement is read-only), the system removes the statement from the KB anyway (even if the data is still in some upstream source). A negative statement is added to the overwrite graph. This negative statement will prevent using a source statement to reintroduce the corresponding statement in KB: The negative statement overwrites the source statement.

4 Enrichers

We describe the general principles of enricher modules. We then describe two specific enrichments: agent matching and event geolocation.

After loading, enricher modules perform inference tasks such as entity resolution, event geolocation, and other knowledge enrichment tasks. We distinguish between two kinds of enrichers. The first kind takes as input the entire current state of the KB and applies to it a set Δ of enrichments (i.e., new statements). For instance, this is the case for the module that performs entity resolution for agents. The second type of enricher works in a differential manner: it takes as input the current state of the KB, and a collection of changes Δ_i that have happened recently. It computes a new collection Δ_{i+1} of enrichments. Intuitively, this allows reacting to changes of a data source. When some Δ_0 is detected (typically by some synchronizer), the system runs a pipeline of enrichers to take these new changes into consideration. For instance, when a new entry is entered in the calendar with an address, a geocoding enricher is called to attempt to locate it. Another enricher will later try to match it with a position in the location history.

We now present two enrichers that have been incorporated into the system.

Agent Matching. The KB keeps information as close to the original data as possible. Thus, the knowledge base will typically contain several entities for the same person, if that person appears with different names or different email addresses. We call such resources *facets* of the same real-world agent. Different facets of the same agent will be linked by the `personal:sameAs` relation. The task of identifying equivalent facets has been intensively studied under different names such as record linkage, entity resolution, or object matching [5]. In our case, we use techniques that are tailored to the context of personal KBs: identifier-based matching and attribute-based matching.

We can match two facets if they have the same value for some particular attribute (such as an email address or a telephone number), which, in some sense, determines the entity. This approach is commonly used for personal information systems (in research and industry) and gives fairly good results for linking, e.g., facets extracted from emails and the ones extracted from contacts. Such a matching may occasionally be incorrect, e.g., when two spouses share a mobile phone or two employees share the same customer relations email address. In our experience, such cases are rare, and we postpone their study to future work.

Two agent facets with the same first and family names have, for instance, a higher probability to represent the same agent than two agent facets with different names, all other attributes held constant. Besides names, other attributes can help determine a matching or not between two agents, `schema:birthDate`, `schema:gender` and `schema:email`.

We tried holistic matching algorithms for graph alignments [24] that we adapted to our setting. The results turned out to be disappointing (see Sec. 5). We believe this is due to the following: (i) almost all agent facets have a `schema:email`, and possibly a `schema:name`, but most of them lack other attributes that are thus almost useless; (ii) names extracted from mails may contain pseudonyms, abbreviations, or lack family names, which reduces matching precision. (iii) we cannot reliably compute name frequency metrics from the knowledge base, since a rare name may appear many times for different email addresses if a person happens to be a friend of the user. Therefore, we developed our own algorithm, *AgentMatch*, which works as follows:

1. We partition `Agents` using the equivalence relation computed by the *Identifier-based Matching* technique previously described.
2. For each `Agent` equivalence class, we compute its corresponding set of names, and, for each name, its number of occurrences (in email messages, etc.).
3. We compute Inverse Document Frequency (IDF) scores, where the documents are the equivalence classes, and the terms are the name occurrences.
4. For each pair of equivalence classes, we compute a numerical similarity between each pair of names using an approximate string distance that finds the best matching of words between the two names, and then compares matching words using another string similarity function (discussed below). The similarity between two names is computed as a weighted mean using the sum of word-IDFs as weights. The best

matching of words corresponds to a maximum weight matching in the bipartite graph of words where weights are computed using the second string similarity function. The similarity (between 0 and 1) between two equivalence classes is computed as a weighted mean of name pair similarity using the product of word occurrences as weights.

5. Pairs for which the similarity is above a certain threshold are considered to correspond to two facets of the same real world entity.

The second similarity function we use is based on the Levenshtein edit-distance, after string normalization (accent removal and lowercasing). In our experiments, we have also tried the Jaro–Winkler distance. For performance reasons, we use 2- or 3-gram-based indexing of words in agent names, and only consider in step (4.) of the process those `Agent` parts with some ratio S of q-grams in common in at least one word. For instance, two `Agent` parts with names "Susan Doe" and "Susane Smith" would be candidates.

Geolocating Events. We discuss how to geolocate events, e.g., how we can detect that Monday's lunch was at "Shana Thai Restaurant, 311 Moffett Boulevard, Mountain View, CA 94043". For this, we first analyze the location history from the user's smartphone to detect places where the user stayed for a prolonged period of time. We then perform some spatio-temporal alignment between such stays and the events in the user's calendar. Finally, we use geocoding to provide location semantics to the events, e.g., a restaurant name and a street address.

Locations in the user's location history can be put into two categories: *stays* and *moves*. *Stays* are locations where the user remained for some period of time (e.g., dinner at a restaurant, gym training, office work), and *moves* are the others. *Moves* usually correspond to locations along a journey from one place to another, but might also correspond to richer outdoor activity (e.g., jogging, sightseeing).

To transform the user's location history into a sequence of stays and moves, we perform time-based spatial clustering [17]. The idea is to create clusters along the time-axis. Locations are sorted by increasing time, and each new location is either added to an existing cluster (that is geographically close and that is not too old), or added to a new cluster. To do so, a location is spatially represented as a two dimensional unimodal normal distribution $\mathcal{N}(\mu, \sigma^2)$. The assumption of a normally distributed error is typical in the field of processing location data. For instance, a cluster of size 1 formed by location point $p = (t, x, y, a)$, where t is the time, a the accuracy, and (x, y) the coordinates, is represented by the distribution $P = \mathcal{N}(\mu_P = (x, y), \sigma_P^2 = a^2)$. When checking whether location p can be added to an existing cluster C represented by distribution Q, the process computes the Hellinger distance [22] between the distribution P and the normal distribution $Q = \mathcal{N}(\mu_Q, \sigma_Q^2)$: $H^2(P, Q) = 1 - \sqrt{\frac{2\sigma_P\sigma_Q}{\sigma_Q^2+\sigma_Q^2}} \exp\left(-\frac{1}{4}\frac{d(\mu_P,\mu_Q)^2}{\sigma_P^2+\sigma_Q^2}\right) \in [0, 1]$, where $d(\mu_P, \mu_Q)$ is the geographical distance between cluster centers. The Hellinger distance takes into account both the accuracy and geographical distance between cluster centers, which allows us to handle outliers no matter the location accuracy. The location is added to C if this distance is below a certain threshold λ, i.e., $H^2(P, Q) \leqslant \lambda^2 < 1$. In our system, we used a threshold of 0.95.

When p is added to cluster C, the resulting cluster is defined with a normal distribution whose expectation is the arithmetic mean of location point centers weighted by the inverse accuracy squared, and whose variance is the harmonic mean of accuracies squared.

A cluster that lasted more than a certain threshold is a candidate for being a stay. A difficulty is that a single location history (e.g., Google Location History) may record locations of different devices, e.g., a telephone and a tablet. The identity of the device may not be recorded. The algorithm understands that two far-away locations, very close in time, must come from different devices. Typically, one of the devices is considered to be stationary, and we try to detect a movement of the other. Another difficulty comes when traveling in high speed trains with poor network connectivity. Location trackers will often give the same location for a few minutes, which leads to the detection of an incorrect stay.

After the extraction of stays using the previous algorithm, the next step is to match these with calendar events. Such a matching turns out to be difficult because: (i) the location of an event (address or geo-coordinates) is often missing; (ii) when present, an address often does not identify a geographical entity, as in "John's home" or "room C110"; (iii) in our experience, starting times are generally reasonable (although a person may be late or early for a meeting) but durations are often not meaningful (around 70% of events in our test datasets were scheduled for 1 hour; among the 1-hour events that we aligned, only 9% lasted between 45 and 75 minutes); (iv) some stays are incorrect.

Because of (i) and (ii), we do not rely much on the location explicitly listed in the user's calendars. We match a stay with an event primarily based on time: the time overlap (or proximity) and the duration. In particular, we match the stay and the event, if the ratio of the overlap duration over the entire stay duration is greater than a threshold θ. As we have seen, event durations are often unreliable because of (iii). Our method still yields reasonable results, because it tolerates errors on the start of the stay for long stays (because of their duration) and for short ones (because calendar events are scheduled usually for at least one hour). If the event has geographical coordinates, we filter out stays that are too far away from that location (i.e., when the distance is greater than δ). We discuss the choice of θ and δ for this process in Sec. 5.

Once stays associated with events, we enrich events with rich place semantics (country, street name, postal code, place name). If an event has an explicit address, we use a *geocoder*. Thymeflow allows using different geocoders, e.g., the Google Maps Geocoding API[1], which returns the geographic coordinates of an address, along with structured place and address data. The enricher only keeps the geocoder's most relevant result and adds its data (geographic coordinates, identifier, street address, etc.) to the location in the knowledge base. For events that do not have an explicit address but that have been matched to a stay, we use the geocoder to transform the geographic coordinates of the stay into a list of nearby places. The most precise result is added as the event location. If the event has both an explicit address and a match with a stay, we call the geocoder on

[1] `https://developers.google.com/maps/documentation`

Table 1: Loading times (minutes) of Angela's dataset, for the creation of 1.6M of triples

	Cold start		Tepid start		Warm start	
	B	N	B	N	B	N
MacBook Air 1.3GHz, 4GB RAM, SSD	28	14	13.0	7.0	0.70	0.67
Desktop PC 3.4GHz, 20GB RAM, SSD	19	10	4.0	2.6	0.22	0.20

this address, while restricting the search to a small area around the stay coordinates.

5 Experiments

In this section, we present the results of our experiments. We used datasets from two real users, whom we call Angela and Barack. Angela's dataset consists of 7,336 emails, 522 calendar events, 204,870 location points, and 124 contacts extracted from Google's email, contact, calendar, and location history services. This corresponds to 1.6M triples in our schema. Barack's dataset consists of 136,301 emails, 3,080 calendar events, 1,229,245 location points, and 582 contacts extracted from the same sources. This corresponds to 10.3M triples.

5.1 KB construction

We measured the loading times (Table 1) of Angela's dataset in different scenarios: (1) a "cold start" from source data on the Internet (using Google API, except for the location history which is not provided by the API and was loaded from a file), (2) a "tepid start" from source data stored in local files, and (3) a "warm start" from a backup file of the knowledge base. Numbers are given with (B) and without (N) loading email bodies for full text search support. In general, loading takes in the order of minutes. Restarting from a backup of the knowledge base takes only seconds.

5.2 Agent matching

We evaluated the precision and recall of the AgentMatch algorithm (Sec. 4) on Barack's dataset. This dataset contains 40,483 `Agent` instances with a total of 25,381 `schema:name` values, of which 17,706 are distinct; it also contains 40,455 `schema:email` values, of which 24,650 are distinct. To compute the precision and recall, we sampled 2,000 pairs of distinct `Agent`s, and asked Barack to assign to each possible pair a ground truth value (true/false).

We tested both Levenshtein and Jaro–Winkler as secondary string distance, with and without IDF term weights. The term q-gram match ratio (S) was set to 0.6. We varied λ so as to maximize the F1 value (see a technical report for details [19]). Our baseline is IdMatch, which matches two contacts iff they have the same email address.

As competitor, we considered PARIS [24], an ontology alignment algorithm that is parametrized by a single threshold. We used string similarity for email addresses, and the name similarity metric used by AgentMatch, except that it is applied to single `Agent`

Table 2: Precision and recall of the Agent Matching task on Barack's dataset, for different parameters of the AgentMatch, IdMatch, PARIS and Google algorithms.

Algorithm	**Similarity**	**IDF**	λ	**Prec.**	**Rec.**	**F1**
IdMatch				1.000	0.430	0.601
Google1				0.995	0.508	0.672
Google2				0.996	0.453	0.623
GoogleId2				0.997	0.625	0.768
GoogleId1				0.996	0.608	0.755
PARIS	Jaro–Winkler	T	0.425	0.829	0.922	0.873
AgentMatch	Levenshtein	F	0.725	0.945	0.904	0.924
AgentMatch	Levenshtein	T	0.775	0.948	0.900	0.923
AgentMatch	Jaro–Winkler	F	0.925	0.988	0.841	0.909
AgentMatch	Jaro–Winkler	T	0.825	0.954	0.945	0.949

instances. PARIS computes the average number of outgoing edges for each relation. Since our dataset contains duplicates, we gave PARIS an advantage by computing these values upfront.

We also considered Googles "Find duplicates" feature. Google was not able to handle more than 27,000 contacts at the same time, and so we had to run it multiple times in batches. Since the final output depends on the order in which contacts were loaded, we present two results, one for which the contacts were supplied sorted by email address (Google1), and another for a random order (Google2). Since Google's algorithm failed to merge contacts that IdMatch did merge, we also tested running IdMatch on Google's output (GoogleId) for both runs. We also tested Mac OS X contact de-duplication feature. However, its result did not contain all the meta data from the original contacts, so that we could not evaluate this feature.

The results are shown in Table 2. As expected, our baseline IdMatch has a perfect precision, but a low recall (43%). Google, likewise, gives preference to precision, but achieves a higher recall than the baseline (50%). The recall improves further if the Google is combined with IdMatch (61%). PARIS, in contrast, favors recall (92%) over precision (83%), and achieves a better F1 value overall. The highest F1-measure (95%) is reached for AgentMatch with the Jaro–Winkler distance for a threshold of 0.825. It has a precision comparable to Google's, and a recall comparable to PARIS'.

5.3 Detecting stays

We evaluated the extraction of stays from the location history on Barack's dataset. We randomly chose 15 days (among 1676), and presented him a Web interface with (1) the raw locations on a map, (2) the stays detected by Thymeflow, and (3) the stays detected by his Google Timeline [13]. Barack was then asked to annotate these stays as "true" or "false", and to report missing stays, based on his memories.

Table 3 shows the resulting precision and recall for each method, with varying stay duration thresholds for Thymeflow. We also show the duplicate ratio #D (number of duplicates over the number of true stays) and the duplicate duration ratio DΘ (duplicate

Table 3: Stay extraction evaluation on Barack's dataset.

Method	θ	#D	DΘ	Prec.	Recall	F1
Thyme	15	33.3 %	0.6 %	91.3 %	98.4 %	94.7 %
Thyme	10	46.0 %	0.9 %	82.9 %	98.4 %	90.0 %
Thyme	5	62.5 %	1.0 %	62.1 %	100.0 %	76.6 %
Google		17.5 %	N/A	87.7 %	89.1 %	88.4 %

duration over the total duration of true stays). Overall, Thymeflow obtains results comparable to Google Timeline for stay extraction, with better precision and recall, but more duplicates.

Matching stays with events. We also evaluated the matching of stays in the user's location history with the events in their calendar. For space reasons, we describe these experiments only briefly here, and refer the reader to our technical report for details [19]. We sampled stays from Angela and Barack's datasets, and produced all possible matchings to events, i.e., all matchings produced by the algorithm whatever the threshold. Angela and Barack were then asked to manually label the matches as correct or incorrect. The matching process relies on two parameters, namely the duration ratio threshold θ and the filtering distance δ. We varied θ and found that a value of 0.2 leads to best F1 values. With this value, we varied δ, and found that the performance improves consistently with larger values. This indicates that filtering stays which are too far from event location coordinates (where available) should not be taken into consideration. With these settings, the matching performs quite well: We achieve a precision and recall of around 70%.

5.4 Geocoding

We evaluated different geocoding enrichers for (event, stay) pairs described in Sec. 4. We used the Google Maps Geocoding API. We considered three different inputs to this API: the event location address attribute (Event), the stay coordinates (Stay), and the event location attribute with the stay coordinates given as a bias (StayEvent). We also considered a more precise version of Event, which produces a result only if the geocoder returns a single unambiguous location (EventSingle). Finally, we devised the method StayEvent+Stay, which returns the StayEvent result if it exists, and the Stay result otherwise.

For each input, the geocoder gave either no result (M), a false result (F), a true place (T), or just a true address (A). For instance, an event occurring in “Hôtel Ritz Paris” is true if the output is for instance “Ritz Paris”, while an output of “15 Place Vendôme, Paris” would count as a true address. For comparison, we also evaluated the place given by Google Timeline [13].

Due to limitations of the API, geocoding from stay coordinates mostly yielded address results (99.2% of the time). To better evaluate these methods, we computed the number of times in which the output was either a true place, or a true address (denoted T|A).

Table 4: Geocoding task on matched (event, stay) pairs in Barack's dataset (in %).

Method	M	F	T	P_T	T|A	$P_{T\|A}$	F1
GoogleTimeline	0	82.8	14.8	14.8	17.2	17.2	29.4
EventSingle	50	40.8	4.0	7.9	9.6	19.0	27.7
Event	26	63.6	4.0	5.4	10.0	13.6	22.9
Stay	0	69.6	0.8	0.8	30.4	30.4	46.6
StayEvent	50	16.4	27.2	54.4	33.6	67.2	57.3
StayEvent|Stay	0	50.0	28.4	28.4	50.0	50.0	66.7

For those methods that did not always return a result, we computed a precision metric $P_{T|A}$ (resp., P_T), that is equal to the ratio of T|A (resp., T) to the number of times a result was returned. We computed a F1-measure based on the $P_{T|A}$ precision, and a recall assimilated to the number of times the geocoder returned a result $(1 - M)$.

The evaluation was performed on 250 randomly picked (stay, event) pairs in Barack's dataset. The results are shown in Table 4. The Google Timeline gave the right place or address only 17.2% of the time. The EventSingle method, likewise, performs poorly, indicating that the places are indeed highly ambiguous. The best precision ($P_{T|A}$ of 67.2%) is obtained by Geocoding with the StayEvent, but this method returns a result only 50.0% of the time. StayEvent+Stay, in contrast, can find the right place 28.4% of the time, and the right place or address 50.0% of the time, which is our best result. We are happy with this performance, considering that around 45% of the event locations were room numbers without mention of a building or place name (i.e., C101).

Use cases. Finally, we briefly illustrate uses of the personal KB with some queries. Our triple store is equipped with a SPARQL 1.1 compliant engine [15] with an optional full text indexing feature based on Apache Lucene. Since the KB unites different data sources, queries can seamlessly span multiple sources and data types. This allows the user (Angela), to ask for instance:

- What are the phone numbers of her birthday party guests? (combining information derived from the contacts and the emails)
- What places did she visit during her last trip to London? (combining geocoding information with stays)
- For each person she meets more than 3 times a week, what are the top 2 places where she usually meets that particular person? (based on her calendar and location history)

Such queries are not supported by current commercial solutions. More examples are provided in the technical report [19] and in a demo article [20].

6 Related Work

This work is motivated by the general concept of personal information management, see, e.g., [1]. The problem of building a knowledge base for querying and managing personal information is not new. Among the first projects in this direction were IRIS [4] and NEPOMUK [14]. These used Semantic Web technologies to exchange data between different applications within a single desktop computer and also provided semantic search facilities for desktop data. However, these projects date from 2005 and 2007, respectively, and much has changed since then. Today, most of our personal information is not stored on an individual computer, but spread across several devices [1]. Our work is different from these projects in three aspects. (i) We do not tackle personal information management by reinventing the user experience for reading/writing emails, managing a calendar, organizing files, etc. (ii) We embrace personal information as being fundamentally distributed and focusing on the need of providing integration on top for creating completely new services (complex query answering, analytics). (iii) While NEPOMUK provides text analysis tools for extracting entities from rich text and linking them with elements of the Knowledge Base, our focus is on enriching existing semi-structured data.

Data matching (also known as record linkage, entity resolution, information integration, or object matching) is extensively utilized in data mining projects and in large-scale information systems by business, public bodies and governments [5]. Example application areas include national census, the health sector, or fraud detection. Recently, contact managers from known vendors have started providing de-duplication tools for finding duplicate contacts and merging them in bulk. However, these tools restrict themselves to contacts present in the user's address book and do not necessarily merge contacts from social networks or emails.

The ubiquity of networked mobile devices able to track users' locations over time has been greatly utilized for estimating traffic and studying mobility patterns in urban areas. Improvements in accuracy and battery efficiency of location technologies have made possible the estimation of user activities and visited places on a daily basis [2, 17, 13]. Most of these studies have mainly exploited sensor data (accelerometer, location, network) and readily available geographic data. A recent study has recognized the importance of fusing location histories with location data for improving the representation of information contained in the user's calendar [18].

Common standards, such as vCard and iCalendar have allowed provider-independent administration of personal information. There is also a proposed standard for mapping vCard content and iCalendars into RDF [16, 6]. While such standards are useful in our context, they do not provide the means to match calenders, emails, and events, as we do. The only set of vocabularies besides `schema.org` which provides a broad coverage of all entities we are dealing with is the OSCAF ontologies [21], which is currently unmaintained.

Commercial providers such as Google and Apple, have arguably come close to our vision of a personal knowledge base, by integrating calendars, emails, address books, allowing smart exchanges between them. Google Now even pro-actively interacts with the user. However, these are proprietary solutions, not under the control of the user, and limited to whatever data is available to the company.

7 Conclusion

The Thymeflow system integrates data from emails, calendars, address books, and location history. It can merge different facets of the same agent, determine prolonged stays in the location history, and align them with events in the calendar. It is available under an open-source software license[2]. People can therefore freely use it, and researchers can build on it. Thymeflow is, to our knowledge, the first open system that allows users to reclaim their personal data from commercial providers, to manage it, to enrich it with spatio-temporal information, and to query it.

Our system can be extended in a number of directions, including incorporating more data sources, extracting semantics from text (in particular email bodies), complex analysis of users' data and behavior. Also, our system could be used proactively to interact with the user, in the style of Apple's Siri, Google's Google Now, Microsoft's Cortana, or Amazon Echo.

Thymeflow is not intended to replace any existing application supporting various kinds of personal data and different functionalities. It provides the integration of the data sources as a knowledge base, and novel functionalities on top of it. We insist on the fact that Thymeflow is meant to remain under the direct control of the user, and fully respect the privacy of the user's data. In particular, the flow of data from Thymeflow to external data sources is fully controlled by the user.

References

[1] Serge Abiteboul, Benjamin André, and Daniel Kaplan. Managing your digital life. *CACM*, 58(5), 2015.

[2] Daniel Ashbrook and Thad Starner. Using GPS to learn significant locations and predict movement across multiple users. *Personal and Ubiquitous Computing*, 7(5), 2003.

[3] Jeen Broekstra, Arjohn Kampman, and Frank Van Harmelen. Sesame: A generic architecture for storing and querying RDF and RDF schema. In *ISWC*, 2002.

[4] Adam Cheyer, Jack Park, and Richard Giuli. IRIS: Integrate. relate. infer. share. Technical report, DTIC Document, 2005.

[5] Peter Christen. *Data matching: concepts and techniques for record linkage, entity resolution, and duplicate detection.* Springer, 2012.

[6] Dan Connolly and Libby Miller. *RDF Calendar - an application of the Resource Description Framework to iCalendar Data.* 2005. http://www.w3.org/TR/rdfcal/.

[7] David H. Crocker. Standard for the format of ARPA internet text messages. RFC 822, IETF, 1982.

[2] https://github.com/thymeflow/thymeflow

[8] Richard Cyganiak, David Wood, and Markus Lanthaler. *RDF 1.1 Concepts and Abstract Syntax.* 2014. `http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/`.

[9] B. Desruisseaux. Internet calendaring and scheduling core object specification (iCalendar). RFC 5545, IETF, 2009.

[10] Facebook. The graph API.

[11] Hector Garcia-Molina, Yannis Papakonstantinou, Dallan Quass, Anand Rajaraman, Yehoshua Sagiv, Jeffrey Ullman, Vasilis Vassalos, and Jennifer Widom. The TSIMMIS approach to mediation: Data models and languages. *Journal of intelligent information systems*, 8(2), 1997.

[12] Paula Gearon, Alexandre Passant, and Axel Polleres. *SPARQL 1.1 Update.* 2013. `https://www.w3.org/TR/sparql11-update/`.

[13] Google. Google maps timeline.

[14] Siegfried Handschuh, Knud Möller, and Tudor Groza. The NEPOMUK project-on the way to the social semantic desktop. In *I-SEMANTICS*, 2007.

[15] Steve Harris, Andy Seaborne, and Eric Prud'hommeaux. *SPARQL 1.1 Query Language.* 2013. `http://www.w3.org/TR/sparql11-query/`.

[16] Renato Iannella and James McKinney. *vCard Ontology - for describing People and Organizations.* 2014. `http://www.w3.org/TR/2014/NOTE-vcard-rdf-20140522/`.

[17] Jong Hee Kang, William Welbourne, Benjamin Stewart, and Gaetano Borriello. Extracting places from traces of locations. In *WMASH*, 2004.

[18] Tom Lovett, Eamonn O'Neill, James Irwin, and David Pollington. The calendar as a sensor: analysis and improvement using data fusion with social networks and location. In *UbiComp*, 2010.

[19] David Montoya, Thomas Pellissier Tanon, Serge Abiteboul, Pierre Senellart, and Fabian Suchanek. Thymeflow, An Open-Source Personal Knowledge Base System. Technical report, 2016.

[20] David Montoya, Thomas Pellissier Tanon, Serge Abiteboul, and Fabian Suchanek. Thymeflow, A Personal Knowledge Base with Spatio-temporal Data. In *Proceedings of the 25th ACM International Conference on Information and Knowledge Management (CIKM'16)*, pages 2477–2480, Indianapolis, Indiana, USA, October 2016. ACM.

[21] Nepomuk Consortium and OSCAF. OSCAF ontologies.

[22] Mikhail S Nikulin. Hellinger distance. *Encyclopedia of Mathematics*, 2001.

[23] S. Perreault. vCard format specification. RFC 6350, IETF, 2011.

[24] Fabian M Suchanek, Serge Abiteboul, and Pierre Senellart. PARIS: Probabilistic alignment of relations, instances, and schema. *PVLDB*, 5(3), 2011.

[25] Denny Vrandečić and Markus Krötzsch. Wikidata: A free collaborative knowledgebase. *CACM*, 57(10), 2014.