Thymeflow, An Open-Source Personal Knowledge Base System

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

Thymeflow, An Open-Source Personal Knowledge Base System

Technical report

David Montoya[1,2,3], Thomas Pellissier Tanon[4], Serge Abiteboul[2,3], Pierre Senellart[5], and Fabian M. Suchanek[6]

[1]Engie Ineo — [2]Inria Paris — [3]ENS Paris-Saclay — [4]ENS Lyon — [5]DI, ENS, PSL Research University — [6]Télécom ParisTech, University Paris-Saclay

December 14, 2016

The typical Internet user has 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. To this end, our system extends the `schema.org` vocabulary and provides a SPARQL interface.

1 Introduction

Typical Internet users today have their data spread over several devices and services. This includes emails, messages, contact lists, calendars, location histories, and many other types of data. However, commercial systems often function as data traps, where it is easy to check in information and difficult to query it. For example, a user may have all her emails stored with an email provider – but she cannot find out who are the people with whom she interacts most frequently. In the same spirit, she may have all her location history on her phone – but she cannot find out at which of her friends' places she spends

the most time. All of this is paradoxical, because the user provides the data but is not allowed to make use of it.

This problem becomes all the more important as more and more of our lives happen in the digital sphere. Seen this way, users are actually giving away part of their life to external data services.

Goal With our work, we aim to put the user back in control of her own data. We introduce a novel framework that integrates personal information from different sources into a single knowledge base (KB) that lives on the user's machine, a machine she controls. Our system replicates data from outside services (such as email, calendar, contacts, GPS services, etc.), 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.

Challenges 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, our integration task spans data of different modalities: if we want to create a coherent user experience, we have to align events in the calendar (temporal information) with GPS traces (location data) and place names.

Contributions 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 [16] and GPS Logger [24]. We believe that such data becomes useful only if it is semantically enriched with events and people in the user's personal space – which is what we do.

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 the alignments based on text (a person mentioned in a calendar entry and a contact), on time (a meeting in calendar and a GPS position) or on 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 resolutions and spatio-temporal alignments.

Results Our system is fully functional and implemented. It can, e.g., provide answers to questions such as:

- Who are the people I have contacted the most in the past month? (requiring alignments of the different email addresses)
- How many times did I go to Alice's place last year ? (requiring an alignment between the contact list and location history)
- Where did I have lunch with Alice last week? (requiring an alignment between the calendar and location history)

The rest of this paper is structured as follows: Section 2 describes our data model and data sources, Section 3 describes the system architecture, Section 4 details our knowledge enrichment processes, Section 5 discusses experimental results, Section 6 the related work, and Section 7 concludes.

2 The data model

In this section, we briefly describe the representation format we use, namely RDF. We then present the schema of the knowledge base, and discuss the mapping of data sources to that schema.

RDF We use the RDF standard [10] for knowledge representation. RDF uses a set I of *URIs* to identify entities. For example, the URI `http://www.wikidata.org/entity/Q76` identifies the entity Barack Obama. To abbreviate URIs, it is common to define *namespace prefixes*. We use `wd` for `http://www.wikidata.org/entity/`, `schema` for `http://schema.org/`, and `rdf` and `rdfs` for the standard name spaces of RDF and RDFS, respectively. With this, the URI of Barack Obama can be written as `wd:Q76`. Let B be a set of *blank nodes*, i.e., identifiers that serve as local existentially quantified variables. Let L be a set of *literals*, i.e., of strings, numbers, and dates – such as `"John Doe"` or `"1990-01-01"`. The set $R = I \cup B \cup L$ is the set of all *resources*. Let P be a subset of I representing *properties*, such as the property `http://schema.org/name`. A *statement* is a triple in $(I \cup B) \times P \times R$. For example, we can state the name of Barack Obama as: $\langle$`wd:Q76`, `schema:name`, `"Obama"`$\rangle$. A *named graph* is a set of statements with an IRI (its name). A *knowledge base* (KB) is a set of named graphs.

Let C be a subset of I that represents *classes*, such as `person` or `city`. The `rdf:type` property says that an entity is an instance of a class, as in $\langle$`wd:Q76`, `rdf:type`, `schema:Person`$\rangle$. The property `rdfs:subClassOf` says that all instances of one class are also instances of another class, as in

$$\langle \texttt{schema:Person}, \texttt{rdfs:subClassOf}, \texttt{schema:Thing} \rangle.$$

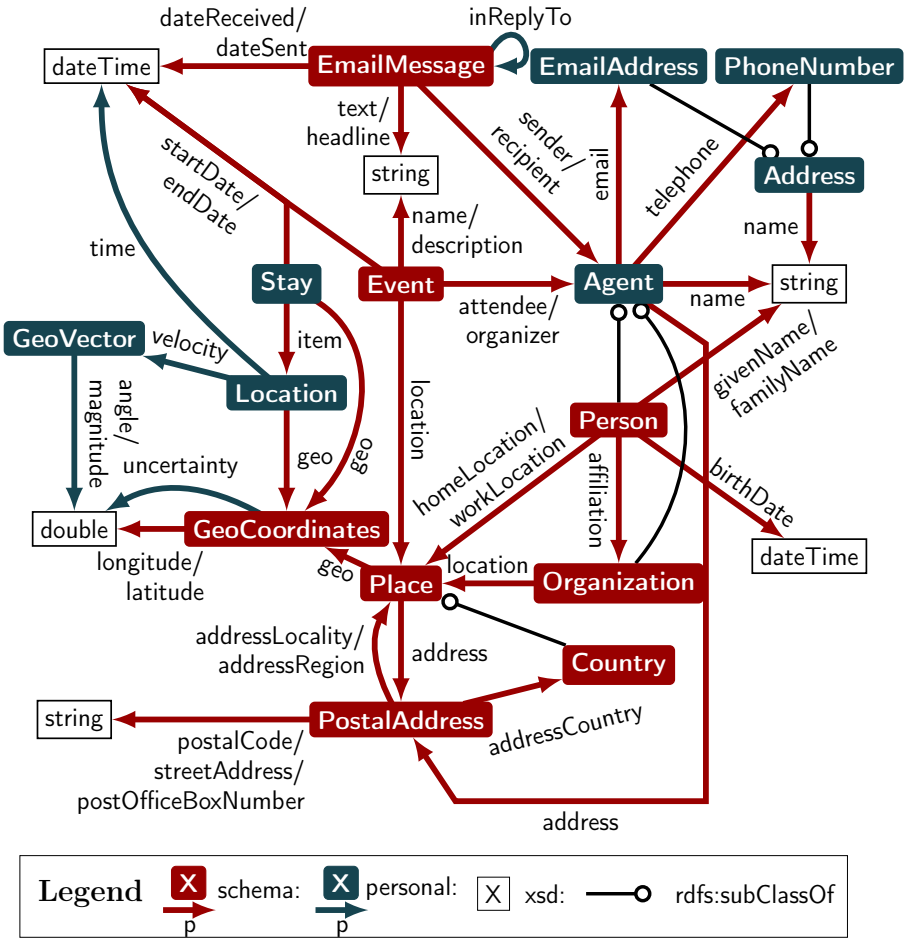

Figure 1: Personal Data Model

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

Figure 1 illustrates a part of our schema. Nodes represent classes, rounded ones are non-literal classes, and an arrow edge with label p from X to Y means that the property 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 Stays, Email-Addresses, and Phone-Numbers. People and Organization classes are aggregated into a `personal:Agent` class.

Emails We treat emails in the RFC 822 format [9]. 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, as e.g., "j4569@gmail.com". Sometimes, email fields contain a name, as in "Jane Doe ⟨ j4569@gmail.com ⟩", which gives us a name triple. Sometimes, the name contains some extra information, as in "Jane - Human Resources ⟨ jane.doe@inria.fr ⟩". We cannot currently extract such additional information. In our model, `personal:Agent` instances extracted from emails with the same 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`.

Contact lists A contact in the vCard format [27] is represented as an instance of `personal:Agent` with properties such as `schema:name`, `schema:familyName`, `schema:email`, `schema:address`. We normalize fields such as telephone numbers, based on a country setting provided by the user as input.

Calendar The iCalendar format [11] can represent events and to-dos. The latter are not supported by all providers, and we do not consider them for now. Events are represented by 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 [12] 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 (GPS/GLONASS), Wi-Fi and Cellular, with varying degrees of accuracy. 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 usually represented by time, longitude, latitude, and horizontal accuracy (the measurement's standard error). These applications store location data in files of various formats, including GPX and KML, but also flat CSV. No standardized protocol exists for managing a location history (i.e. with synchronization capabilities). In our system, we use the Google Location History format, in JSON, as most users with a Google account can easily export their history in this format. In our model, a given point in the user's location history is represented by a resource of type `personal:Location` with two main properties, one of type `schema:geo` (for geographic coordinates with accuracy) and one for time, namely `personal:time`.

3 The system

Mediation vs Warehouse One could first see our personal knowledge base as a *view* defined over the personal information sources. The user would query this view in a mediation style [13], loading the data only on demand. However, accessing, analyzing and integrating these data sources on the fly are expensive tasks. In some cases, the integration may require iterating through the entire data source, which can be prohibitively costly in the case of emails. Also, we want to avoid having to redo the same inferences. For instance, suppose that we had automatically inferred that an event in the calendar took place at a certain location. We do not want to infer this piece of information again. Our approach therefore loads the data sources into a persistent store, in a style that enriches the data warehouse.

System Our system is a Scala program that the user installs. The user provides the system with a list of data sources (such as email accounts, calendars, or address books), together with authorizations to access them (such as tokens or passwords). The system accesses the data sources (as the user would), 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 Sesame based triple store [4]. Since the knowledge base is persistent, we can stop and restart the system without losing information.

Architecture One of the main challenges in the creation of a personal KB is the temporal factor: data sources may change, and these updates have to be reflected in the KB. These changes can happen during the initial load time, while the system is asleep, or after some inferences have already been computed. To address these dynamics, our system uses software modules called *synchronizers* and *enrichers*. Figure 2 shows the synchronizers $S_1, \ldots, S_n$ on the left, and the enrichers $E_1, \ldots, E_p$ in the center. Synchronizers are responsible for accessing the data sources. Enrichers are responsible for inferring new statements, such as alignments between entities obtained by entity resolution.

These modules are scheduled dynamically. For example, some modules 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 (e.g., daily) for particularly costly alignment processes. Generally, when a synchronizer detects a change in a source, it triggers the execution of a pipeline of enricher modules, as shown in Figure 2.

Enrichers can also use knowledge from external data sources, such as Wikidata [30], Yago [29], or OpenStreetMap, as shown in the figure.

Loading Synchronizer modules are responsible for retrieving new data from a data source. We have designed adapters that transform items various the data sources into our schema, as discussed in Section 2. For each data source that has been updated, the

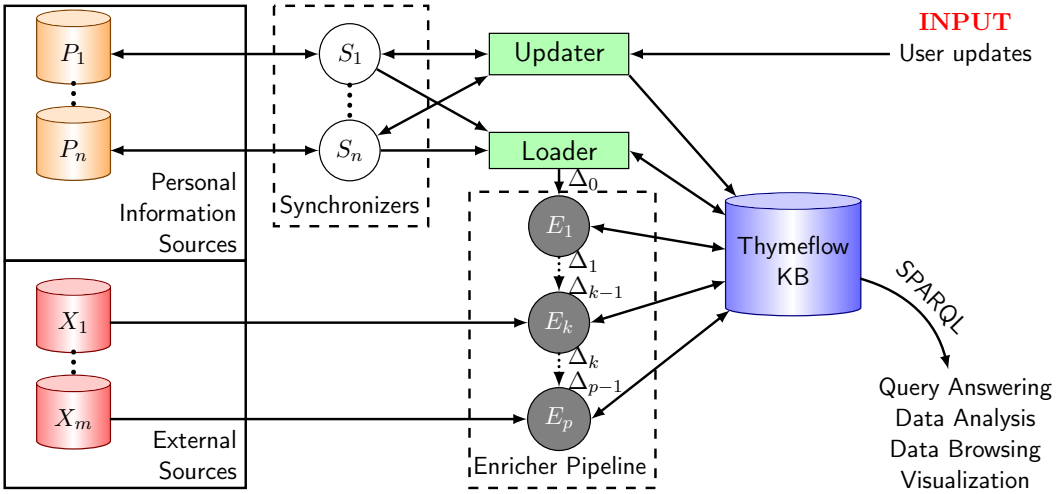

Figure 2: System architecture

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 Loader is responsible for persisting these changes in the KB and for triggering the pipeline of enrichers.

Provenance For each newly obtained piece of information, our KB records the provenance. For synchronizers, this is the data source, and for enrichers it can be, for example, the Agent Matching algorithm described in Section 4.2. 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 for answering queries such as "What are the meetings next Monday that are recorded in my work calendar?".

Pushing to the source Finally, the system allows the propagation of information from the KB to the data sources. These can be either 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 have to translate from the structure and terminology of the KB 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 mobile phone's contact list.

Data from some sources are not updatable from the KB, for example emails. By contrast, CardDAV contacts can be updated. The user can update the KB by inserting or deleting knowledge statements. Such updates to the KB are specified in the SPARQL Update language [14]. The system propagates these to the sources. The semantics of propagation is as follows:

If a target is specified, the system tries to propagate the new information to that specific source. If the operation fails, the error is reported to the user.

Now consider the case when the user does not specify explicitly a target. For insertions, the system tries to find all applicable targets for the operation. All sources where the subject of the statement is already described (i.e. all sources that contain a statement with the same subject) are selected. If no source is able to register an insertion, the system performs the insertion in a special graph in the KB, the *Overwrite Graph*.

For deletion, all sources that contain this triple are selected. Then the system attempts to perform the deletion on the selected sources. If one source fails to perform a deletion (e.g. a source containing the statement is read-only), the system removes the statement from the KB anyway (even if the data is still in some upstream source) and adds a negated statement to the Overwrite Graph to remember that this statement should not be added again to the KB. The negated statement has the same subject and object as the original statement but uses a negated version of the predicate. It should be seen as overwriting the source statement. The user can also specify the deletion of enricher matchings (either a specific one with a target, or any). The Overwrite Graph records these with negative match statements. This will prevent from "rederiving" these matchings.

4 Enrichers

In this section, we describe the general principles of enricher modules (Section 4.1), and then two specific enrichments: agent matching (Section 4.2) and event geolocation (Section 4.3).

4.1 Purpose

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 the enrichers that have already been incorporated into the system.

4.2 Agent Matching

Facets Our schema 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 and/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 [7]. In our case, we use techniques that are tailored to the context of personal KBs: identifier-based matching and attribute-based matching.

Identifier-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. We are aware that 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.

Attribute-based Matching Identifier-based matching cannot detect that john.doe@gmail.com and john.doe@hotmail.com are two facets of the same person. In some cases, other attributes can help. For instance, two agent facets with the same name have a higher probability to represent the same agent than two agent facets with different names, all other attributes held constant. In our schema, the following attributes could help the matching process: `schema:name`, `schema:givenName`, `schema:familyName`, `schema:birthDate`, `schema:gender` and `schema:email`.

We tried holistic matching algorithms suited for instance alignment between two graphs [28], which we adapted for our setting. The results turned out to be disappointing (cf. experiments in Section 5.2). We believe this is due to the following:

- Schema heterogeneity: for instance, almost all agent facets have a `schema:email`, and possibly a `schema:name`, but most of them lack `schema:givenName`, `schema:familyName`, `schema:gender` or `schema:birthDate`.
- Imprecision: names extracted from mails may contain pseudonyms, abbreviations, or may lack family names, and this reduces matching precision. Some attributes

may appear multiple times with different values.

- Lack of good priors: we cannot reliably compute name frequency metrics from the knowledge base, since it may be the case that a rare name appears 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 `Agent`s 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 an Inverse Document Frequency (IDF) table, 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 their respective 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. Finally, 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. We keep the pairs for which the similarity is above a certain threshold. The two facets of these pairs are considered to correspond to 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 2/3-gram-based indexing of words in agent names, and only consider in Step (4.) of the above 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. We evaluate the performance of these agent matching techniques in our experiments.

4.3 Geolocating Events

In this section, we discuss how we geolocate events, i.e., how we can detect for example 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.

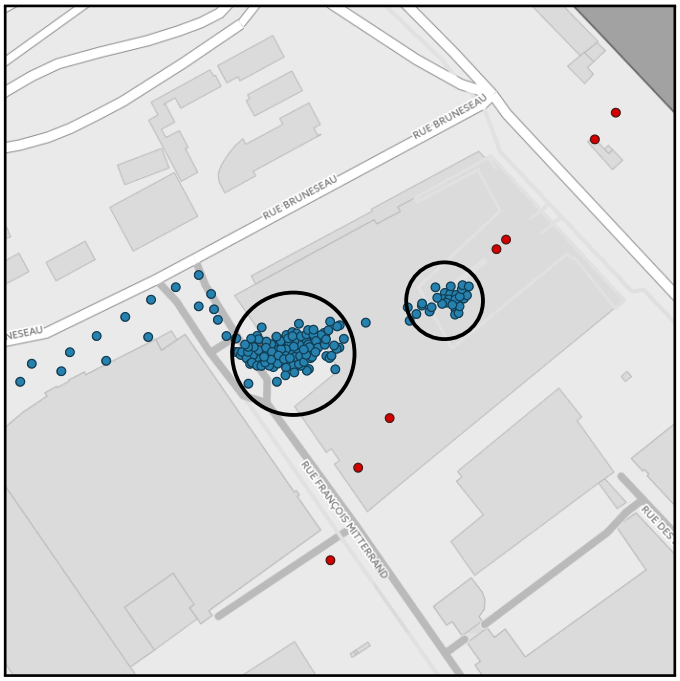

Figure 3: Two clusters of *stays* (blue points inside black circles) within the same building. Red points are outliers. The other points are *moves*.

Detecting stays 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 other ones. *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). Figure 3 illustrates two *stay* clusters located inside the same building.

To transform the user's location history into a sequence of stays and moves, we perform time-based spatial clustering [21]. 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 [26] between the distribution P and the normal distribution $Q = \mathcal{N}(\mu_Q, \sigma_Q^2)$:

$$H^2(P, Q) = 1 - \sqrt{\frac{2\sigma_P\sigma_Q}{\sigma_Q^2 + \sigma_Q^2}}\, e^{-\frac{1}{4}\frac{d(\mu_P,\mu_Q)^2}{\sigma_P^2+\sigma_Q^2}} \in [0, 1],$$

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., $H^2(P,Q) \leq \lambda^2 < 1$. 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. For instance, if a cluster C is formed by locations $\{p_1, \ldots, p_n\}$, where $p_i = (t_i, x_i, y_i, a_i)$, then C is defined with distribution $\mathcal{N}(\mu, \sigma^2)$ where μ is the weighted arithmetic mean of location centers (x_i, y_i) weighted by their inverse accuracy squared a_i^{-2}, and the variance σ^2 is the harmonic mean of location accuracies squared a_i^{-2}.

$$\mu = \sum_{i=1}^{n} \frac{(x_i, y_i)}{a_i^2} \left(\sum_{i=1}^{n} \frac{1}{a_i^2} \right)^{-1} \qquad \sigma^2 = \left(\sum_{i=1}^{n} \frac{1}{a_i^2} \right)^{-1}$$

The coordinates are assumed to have been projected to an Euclidian plane locally approximating distances and angles on Earth around cluster points. If $n = 1$, then $\mu = (x_1, y_i)$ and $\sigma^2 = a_1^2$, which corresponds to the definition of a cluster of size 1 from Section 4.3.

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.

Matching stays with events 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:

1. The location (an address and/or geo-coordinates) is often missing;
2. When present, an address often does not identify a geographical entity, as in "John's home" or "room C110".
3. In our experience, starting times are generally reasonable (although a person may be late or early for a meeting). However, the duration is often not meaningful. For instance, around 70% of events in our test datasets were scheduled for 1 hour. However, among the 1-hour events that we aligned, only 9% lasted between 45 and 75 minutes.

4. Some stays are incorrect.

Because of (1) and (2), 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 (3). 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 our experimental evaluation in Section 5.4.

Geocoding event addresses Once we have associated stays with events, we want to attach events with rich place semantics (country, street name, postal code, place name). If an event has an explicit address, we use a *geocoder*. The system allows using different geocoders, including a combination of Google Places and Google Maps Geocoding APIs[1], as well as Photon[2], for use with OpenStreetMap data. Such a geocoder is given a raw address or place name (such as "Stanford"), and returns the geographic coordinates of matching places, 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.

Geocoding events using matched stays 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 to the location of the event. If the event has both an explicit address and a match with a stay, we call the geocoder on 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 users, Angela and Barack (actual names changed for privacy). 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.

[1] `https://developers.google.com/maps/documentation`
[2] `https://photon.komoot.de`

	load from				restart from	
	Internet		disk files		KB backup	
w/ email bodies & full-text search	yes	no	yes	no	yes	no
MacBook Air 2013 Intel i5-4250U 2-core 1.3GHz 4GB RAM, SSD	28	14	13	7.0	0.70	0.67
Desktop PC Intel i7-2600k 4-core 3.4GHz 20GB RAM, SSD	19	10	4.0	2.6	0.22	0.20

Table 1: Loading times in minutes of Angela's dataset, leading to the creation of 1.6M of triples

5.1 KB Construction

We measured the loading times (Table 1) of Angela's dataset in different scenarios: (1) 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) source data stored in local files (disk), and (3) restarting the system from a backup file of the knowledge base. Numbers are given with and without 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 (Section 4.2) 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). Since non-matching pairs outnumber matching pairs considerably, we performed stratified sampling by partitioning the dataset based on the matches returned by the AgentMatch algorithm for threshold values between 0 and 1, in steps of 0.05. Figure 4 shows the distribution of `Agent` classes by number of distinct emails within the class for classes output by AgentMatch. The identifier-based matching baseline (further simply called IdMatch) is able to reduce the number of distinct agents from 40,483 to 24,677 (61%), while AgentMatch, for a threshold of 0.825, reduces it to 21,603 (53%).

We evaluated four different versions of AgentMatch. 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. For each version of AgentMatch, Table 2 presents the value of the threshold λ for which the F1-measure is maximal. Precision decreases while recall increases for decreasing threshold values (Figure 5).

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

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.

For comparison, we also considered Google Contacts's "Find duplicates", and PARIS [28], an ontology alignment algorithm that is parametrized by a single threshold. We also tested Mac OS X contact deduplication feature, and while it was able to to identify 29,408 distinct agents, their merged contacts did not fully include all metadata from the original ones, which prevented its used for proper evaluation.

Google was not able to handle more than 27,000 contacts at the same time, and could not detect all duplicates at once, so we had to run it multiple times in batches until convergence. However, we noticed the final output was dependent on the order in which contacts were ordered, and present two results, one for which the contacts were supplied sorted by email address (Google1), and another for which the contacts were supplied in a random order (Google2). Since we noticed that Google's algorithm failed to merge contacts that IdMatch did merge, we also tested running IdMatch on Google's output (GoogleId) for both runs. For PARIS, we used string similarity for email addresses, and the name similarity metric used by AgentMatch, except that it is applied to single `Agent` 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 on the output of AgentMatch. Finally, we also show the performance of the parameter-free IdMatch baseline.

Our experiments (Table 2) show that the highest F1-measure is reached for AgentMatch with Jaro–Winkler distance for a threshold of 0.825, for which AgentMatch has a precision of 0.954, a recall of 0.945, and an F1-measure of 0.949. It also out-performs PARIS by a large margin – for reasons that we discussed in Section 4.2. GoogleId and IdMatch's algorithm favor precision, reaching 0.997 and 1.000 respectively, with a recall of 0.6250 and an F1-measure of 0.7683 for GoogleId, while IdMatch only has a recall of 0.430 and an F1-measure of 0.601. However, by changing AgentMatch's threshold, one can reach similar precision levels for higher recalls. For instance, AgentMatch with the Jaro–Winkler distance and IDF weights has a precision of 0.992, a recall of 0.810 and a F1-measure of 0.892 for a threshold of 0.975.

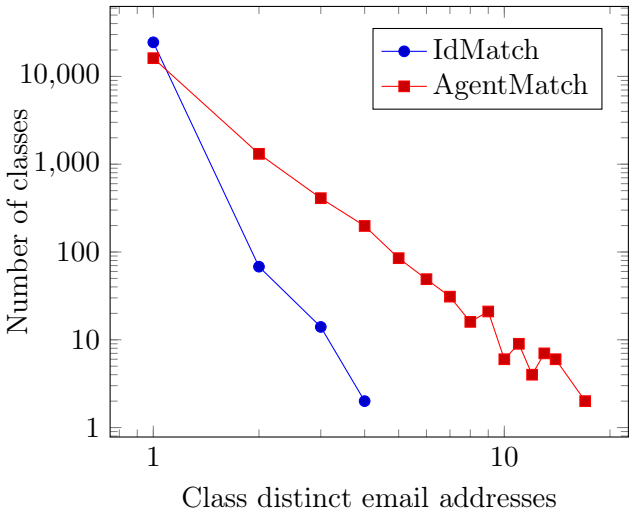

Figure 4: Distribution of `Agent` classes by number of distinct email addresses for classes generated by the best run of AgentMatch on Barack's dataset and classes generated by IdMatch.

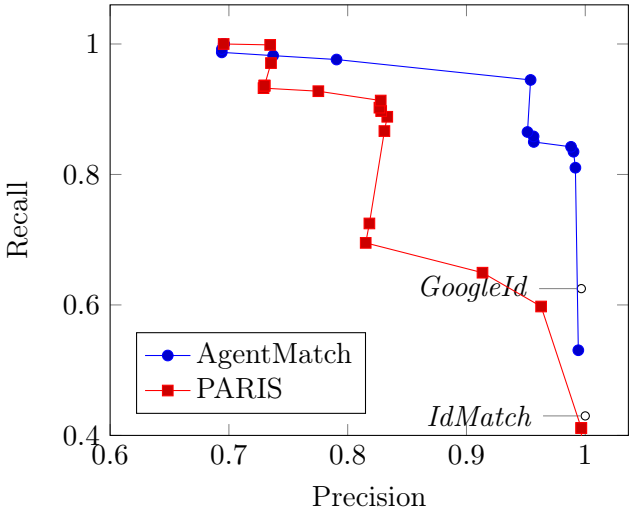

Figure 5: Precision and recall of the Agent Matching task on Barack's dataset, for different threshold values. IdMatch and GoogleId are given as a reference. Precision decreases while recall increases for decreasing threshold values.

Method	#D	DΘ	Prec.	Recall	F1
Thyme15	33.3 %	0.6 %	91.3 %	98.4 %	94.7 %
Thyme10	46.0 %	0.9 %	82.9 %	98.4 %	90.0 %
Thyme5	62.5 %	1.0 %	62.1 %	100.0 %	76.6 %
Google	17.5 %	N/A	87.7 %	89.1 %	88.4 %

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

5.3 Detecting Stays

We evaluated the extraction of stays from the location history on Barack's dataset. Barack annotated 15 randomly chosen days (among 1676) in his location history. For each day, Barack was presented with an interface showing the raw locations of each day on a map, as well as the output of Thymeflow's stay extraction algorithm, for which he varied the stay duration threshold (5, 10, and 15 minutes). For each day, Barack counted the number of stays he thought were true, false, as well as missing. He also evaluated the stays extracted by his Google Timeline [16]. The exact definition of an *actual stay* was left to Barack, and the reliability of his annotations were dependent on his recollection. In total, Barack found 64 *actual stays*. Among the true stays output by each method, some appeared multiple times. Consequently, we counted the number of true stay duplicates and their resulting move duration. For instance, an *actual stay* of 2 hours appearing as two stays of 29m and 88m, with a short move of 3 minutes in-between would count as 1 true stay, 1 duplicate and 3 minutes of duplicate duration. Table 3 shows the resulting precision and recall for each method, as well as the duplicate ratio #D (number of duplicates over the number of true stays), and duplicate duration ratio DΘ (duplicate 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 a tendency to produce duplicates.

5.4 Matching Stays with Events

We evaluated the matching of stays in the user's location history with calendar events on Angela and Barack's datasets. On Angela's dataset, we considered data for one year. This year consists of 129 events and 11,287 stays (as detected by our system) with an average duration of 44 minutes. Candidate matchings, that is, the set of all matchings outputted by the algorithm whatever the chosen thresholds, were manually labeled by Angela. For Barack, we ran the matching algorithm on the full dataset (4.5 years), and Barack labeled the candidate matchings of 216 events picked uniformly at random. Barack's dataset consists of 3,667 events and 49,301 stays totaling 3,105 candidate matchings. The process of matching stays with calendar events relies on two parameters, namely the duration ratio threshold θ and the filtering distance δ (cf. Section 4.3). Figure 6 shows how precision and recall vary depending on the duration ratio θ for a filtering distance set to infinity. It shows some sensibility to θ; the best results are obtained for a value of around 0.15. With

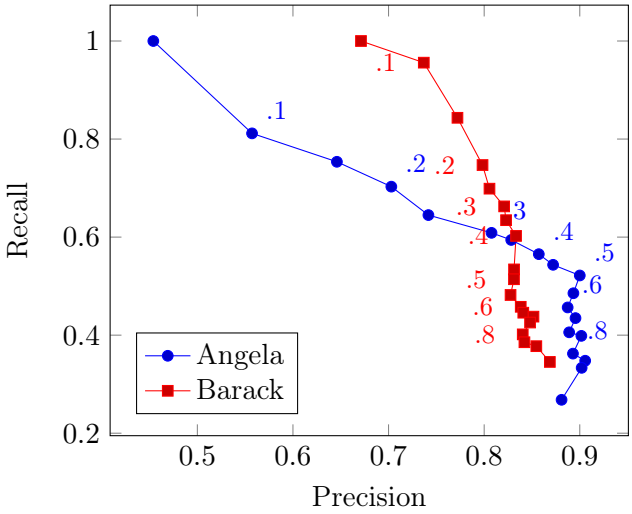

Figure 6: Precision and recall of matching stays with events for different values of the duration ratio θ for a filtering distance δ set to infinity.

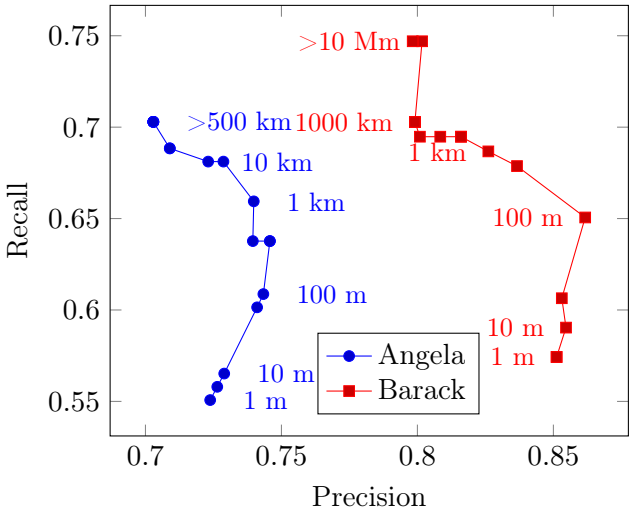

Figure 7: Precision and recall of matching stays with events for different values of the filtering distance θ for a duration ratio threshold θ set to 0.2.

θ set to 0.2, Figure 7 shows the impact of the filtering distance δ. The performance slowly increases with increasing values of δ. This indicates that filtering stays which are too far from event location coordinates (where available) should not be taken into consideration and we have disabled this filter in the final version of this matching implementation (or equivalently, set δ to infinity). In general, the matching performs quite well: We achieve a precision and recall of around 70%.

Method	M	F	T	P_T	T|A	$P_{T|A}$	F1
Event	26	63.6	4.0	5.4	10.0	13.6	22.9
EventSingle	50	40.8	4.0	7.9	9.6	19.0	27.7
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
GoogleTimeline	0	82.8	14.8	14.8	17.2	17.2	29.4

Table 4: Results from the evaluation of the geocoding task on matched (event, stay) pairs in Barack's dataset (in %).

5.5 Geocoding

We experimented with the different geocoding enrichers described in Section 4.3 on Barack's dataset. On this dataset, we evaluated the performance of geocoding matched (event, stay) pairs on Barack's dataset from three different inputs: (i). from the event location address attribute (Event), (ii). from the stay coordinates (Stay), (iii). from the event location attribute with the stay coordinates given as a bias (StayEvent). For this task, we used the Google Places and Maps Geocoding API. 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 be qualified as a true address. For comparison, we also evaluated the place given by Barack's Google Timeline [16]. 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). 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 pairs, and the results are presented in Table 4. We notice that geocoding using the StayEvent method yields the best precision ($P_{T|A}$ of 67.2%), but only returns a result 50.0% of the time. To cope with that, we consider a fourth output, termed StayEvent|Stay, which yields the StayEvent result if it exists, and the Stay result otherwise. This fourth method always returned a result, and gave the right place 28.4% of the time, and the right place or address 50.0% of the time, which is our best result. This is an OK result considering that around 45% of the event locations were room numbers without mention of a building or place name (i.e., C101). For comparison, Google Timeline gave the right place or address 17.2% of the time. To showcase the ambiguity present in an event's location, we also evaluated the output of EventSingle, which was equivalent to Event, except that it kept the most relevant result only if the geocoder's a single result.

```
PREFIX schema: <http://schema.org/>
PREFIX personal: <http://thymeflow.com/personal#>

SELECT ?attendeeTelephone
WHERE {
   ?event a schema:Event ;
      schema:name "Angela's 25th Birthday Party" ;
      schema:attendee/personal:sameAs*
         /schema:telephone
         /schema:name ?attendeeTelephone .
} GROUP BY ?attendeeTelephone
```

Figure 8: A query to retrieve the telephone numbers of attendees of "Angela's 25th Birthday Party" Facebook event.

5.6 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 [19] with an optional full text indexing feature based on Apache Lucene. This allows the user (Angela) to ask, e.g.:

- What are the telephone numbers of her birthday party guests? (So she can send them a last-minute message.)
- What are the places she visited during her previous trip to London? (So she does not go there a second time.)
- What are the latest emails sent by participants of the "Financial Restructuring" working group? (So she quickly reviews them to be ready for this afternoon's meeting.)

Since the KB unites different data sources, queries seamlessly span multiple sources and data types. For instance, the first question in the previous list can ask for the telephone numbers of participants of a Facebook event. However, for multiple reasons, Angela's friends might not have provided a telephone number within the Facebook platform, or these might not be accessible. Since Angela also connected her personal address book into the KB, she can exploit matches found by the system between her Facebook friends and contacts in her personal address book whose telephone numbers are known. For retrieving such matches, we use the `personal:sameAs` relation (Figure 8).

The second question can be answered by retrieving all places ever visited by Angela, and filter on an area around London. Alternatively, she can first retrieve the "London Summer 2015" event from her calendar, and filter stays whose timespan intersects the event's (Figure 9). This way, assuming instead that Angela had been to London several times in the past and she was trying to recall the location of this great pub she went to

on her Summer 2015 trip, this filter on the time-period would instead allow her to more easily find it.

For the third question, Angela uses the KB's full-text search capabilities to first find events whose name partially matches "Financial Restructuring", grabs the list of their attendees, and finally outputs the latest 100 messages that they sent, regardless of the address they used (Figure 10).

Finally, the user can also perform analytics such as:

- Who does she most frequently communicate with? (Taking into account that some people have several email addresses.)
- What are the places where she usually meets one particular person (based on her calendar)?
- What are the most used email providers used by my contacts?
- Who are the people who have sent her the most emails in this particular email folder?

We expect to use these analytics to allow Angela to better present her data, so that she can keep detailed summaries of her interactions with each of her contacts (places, events, communications), instead of only presenting static data (e.g, the contact's telephone number and email address).

```
PREFIX schema: <http://schema.org/>
PREFIX personal: <http://thymeflow.com/personal#>
SELECT ?longitude ?latitude WHERE {
   ?event a schema:Event ;
      schema:name "London Summer 2015" ;
      schema:startDate ?eventStartDate ;
      schema:endDate ?eventEndDate .
   ?stay a personal:Stay ;
      schema:geo [
         schema:longitude ?longitude ;
         schema:latitude ?latitude
      ] ;
      schema:startDate ?stayStartDate ;
      schema:endDate ?stayEndDate .
   FILTER( ?stayStartDate <= ?eventEndDate &&
      ?stayEndDate >= ?eventStartDate)
}
```

Figure 9: A query to show the places visited in London's 2015 trip on a map.

User updates The user can use the source synchronization facilities to enrich her personal address book with knowledge inferred by the system. For instance, she can add to each contact in her vCard address book the email addresses found in other facets matched to this contact (see Figure 11 for an example query).

6 Related Work

This work is motivated by the general concept of personal information management, see e.g. [1].

Personal Knowledge Bases 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 [6] and NEPOMUK [18]. 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

```
PREFIX schema: <http://schema.org/>
PREFIX personal: <http://thymeflow.com/personal#>
PREFIX search: <http://www.openrdf.org/contrib/lucenesail#>
SELECT ?messageHeadline WHERE {
   ?event a schema:Event ;
      schema:attendee ?attendee ;
      search:matches [
         search:query "Financial Restructuring" ;
         search:property schema:name
      ] .
   ?message a schema:Message ;
      schema:sender/personal:sameAs* ?attendee ;
      schema:headline ?messageHeadline ;
      schema:dateSent ?dateSent .
} ORDER BY DESC(?dateSent) LIMIT 100
```

Figure 10: The latest 100 messages sent by a participant of the "Financial Restructuring" meetings.

```
PREFIX schema: <http://schema.org/>
PREFIX personal: <http://thymeflow.com/personal#>
INSERT {
   GRAPH ?provenance {
      ?agent schema:email ?email
   }
} WHERE {
   GRAPH ?provenance {
      ?agent a personal:Agent ;
         schema:name ?name .
   }
   ?provenance a personal:ContactsSource ;
      personal:account/schema:name
         "angela@gmail.com" .
   ?agent personal:sameAs+/schema:email ?email .
   FILTER NOT EXISTS {
      ?agent schema:email ?email .
   }
}
```

Figure 11: A query that adds to each contact in Angela's Google account the email addresses found on matched agents.

linking them with elements of the Knowledge Base, our first focus is on enriching existing semi-structured data, which improves the quality of data for use by other services.

Information Integration 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 [7]. Example application areas include national census, the health sector, fraud detection, online shopping and genealogy. 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.

Location History and Calendar 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 [3, 16, 21, 22, 31]. Most of these studies have mainly exploited sensor data (accelerometer, location, network) and readily available geographic data. Few of them, however, have exploited the user's calendar and other available user data for

creating richer and more semantic activity histories. Recently, a study has recognized the importance of fusing location histories with location data for improving the representation of information contained in the user's calendar: e.g. for distinguishing genuine real-world events from reminders [23].

Open Standards Common standards, such as vCards and iCalendar, have advanced the state of the art by allowing provider-independent administration of personal information. There is also a proposed standard for mapping vCard content and iCalendars into RDF [8, 20]. 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 [25]. But their development was stopped in 2013 and they are not maintained anymore, contrary to `schema.org` which is actively supported by companies like Google and widely used on the web [17].

Email Analysis Several commercial applications and academics have taken to analyzing the content of email. For example, they can recognize simple patterns such as salutations, signatures, and dates [5]. Apple Mail and Gmail can find dates and suggest updates to the user's calendar. Apple Mail can also identify contact information and suggest an update to the user's address book [2]. Semantic data attached to email content in the form of JSON-LD/Microdata annotations can provide information about flight, hotel, train, bus and restaurant reservations [15]. Again other services (such as TripIt or Wipolo) can parse the content of emails to extract travel data and construct a trip schedule. All of these approaches are orthogonal to our work: Such technology can act as enrichment modules in our system. Our work as a whole aims to construct a coherent knowledge base on top of these.

Commercial Solutions Some commercial providers, such as Google and Apple ecosystems, have arguably come quite close to our vision of a personal knowledge base. They integrate calendars, emails, and address books, and allow smart exchanges between them. Google Now even pro-actively interacts with the user. However, these are closed-source and do not allow the scientific community to build on their technology.

7 Conclusion

We have presented our work on building and enriching a personal knowledge base. Our system integrates data from emails, calendars, address books, and the 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,[3] so that people can freely use it, and researchers can build on it.

Our system can be extended in a number of directions, including

[3] `https://github.com/thymeflow/thymeflow`

- Incorporating more data: In particular, we want to include the user's Web search history, more social network data, e.g., from TripAdvisor, and financial transactions.
- Extracting semantics from text: So far, we simply index text. We could, for instance, find in the body of an email the date and location of a meeting that has been planned and perhaps the name of participants (listed or not as recipients of the mail).
- Exploiting this knowledge: As illustrated in the paper, our system can be used for querying. One could consider using it for other purposes such as complex analysis of users data and behavior.
- Improving the user interface: We would like to allow for a simpler query language, maybe even natural language, to open our tool to non-technical users.

Clearly, 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. However, we would like to insist on the fact that Thymeflow is external to the data sources. Its code is open. The system is meant to remain under the direct control of the user, and fully respect the privacy of the user's data.

References

[1] S. Abiteboul, B. André, and D. Kaplan. "Managing your digital life". *CACM* (2015).

[2] Apple. *Mail: Add events, contacts, and more from messages*. URL: `https://support.apple.com/kb/PH22304` (visited on 04/18/2016).

[3] D. Ashbrook and T. Starner. "Using GPS to learn significant locations and predict movement across multiple users". *Personal and Ubiquitous Computing* (2003).

[4] J. Broekstra, A. Kampman, and F. Van Harmelen. "Sesame: A generic architecture for storing and querying RDF and RDF schema". In: *ISWC*. 2002.

[5] V. R. Carvalho and W. W. Cohen. "Learning to extract signature and reply lines from email". In: *CEAS*. 2004.

[6] A. Cheyer, J. Park, and R. Giuli. *IRIS: Integrate. Relate. Infer. Share.* Tech. rep. DTIC Document, 2005.

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

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

[9] D. H. Crocker. *Standard for the format of ARPA Internet text messages*. RFC 822. IETF, 1982.

[10] R. Cyganiak, D. Wood, and M. Lanthaler. *RDF 1.1 Concepts and Abstract Syntax*. `http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/`. 2014.

[11] B. Desruisseaux. *Internet Calendaring and Scheduling Core Object Specification (iCalendar)*. RFC 5545. IETF, 2009.

[12] Facebook. *The Graph API*. URL: `https://developers.facebook.com/docs/graph-api/` (visited on 09/01/2016).

[13] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, V. Vassalos, and J. Widom. "The TSIMMIS approach to mediation: Data models and languages". *Journal of intelligent information systems* (1997).

[14] P. Gearon, A. Passant, and A. Polleres. *SPARQL 1.1 Update*. `https://www.w3.org/TR/sparql11-update/`. 2013.

[15] Google. *Email Markup*. URL: `https://developers.google.com/gmail/markup/` (visited on 04/18/2016).

[16] Google. *Google Maps Timeline*. URL: `https://www.google.fr/maps/timeline` (visited on 09/11/2016).

[17] R. Guha, D. Brickley, and S. Macbeth. "Schema.org: Evolution of structured data on the web". *CACM* (2016).

[18] S. Handschuh, K. Möller, and T. Groza. "The NEPOMUK project-on the way to the social semantic desktop". In: *I-SEMANTICS*. 2007.

[19] S. Harris, A. Seaborne, and E. Prud'hommeaux. *SPARQL 1.1 Query Language*. `http://www.w3.org/TR/sparql11-query/`. 2013.

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

[21] J. H. Kang, W. Welbourne, B. Stewart, and G. Borriello. "Extracting Places from Traces of Locations". In: *WMASH*. 2004.

[22] L. Liao. "Location-based activity recognition". PhD thesis. U. Washington, 2006.

[23] T. Lovett, E. O'Neill, J. Irwin, and D. Pollington. "The calendar as a sensor: analysis and improvement using data fusion with social networks and location". In: *UbiComp*. 2010.

[24] Mendhak. *GPS Logger for Android*. URL: `https://play.google.com/store/apps/details?id=com.mendhak.gpslogger` (visited on 04/18/2016).

[25] Nepomuk Consortium and OSCAF. *OSCAF Ontologies*. URL: `http://www.semanticdesktop.org/ontologies/` (visited on 05/15/2016).

[26] M. S. Nikulin. "Hellinger distance". *Encyclopedia of Mathematics* (2001).

[27] S. Perreault. *vCard Format Specification*. RFC 6350. IETF, 2011.

[28] F. M. Suchanek, S. Abiteboul, and P. Senellart. "PARIS: Probabilistic alignment of relations, instances, and schema". *PVLDB* (2011).

[29] F. M. Suchanek, G. Kasneci, and G. Weikum. "Yago: a core of semantic knowledge". In: *WWW*. 2007.

[30] D. Vrandečić and M. Krötzsch. “Wikidata: A Free Collaborative Knowledgebase”. *CACM* (2014).

[31] Y. Ye, Y. Zheng, Y. Chen, J. Feng, and X. Xie. “Mining individual life pattern based on location history”. In: *MDM*. 2009.