Epistenet: Facilitating Programmatic Access & Processing of Semantically Related Mobile Personal Data

Sauvik Das
Carnegie Mellon University
sauvik@cmu.edu

Jason Wiese
University of Utah
wiese@cs.utah.edu

Jason I. Hong
Carnegie Mellon University
jasonh@cs.cmu.edu

ABSTRACT

Effective use of personal data is a core utility of modern smartphones. On Android, several challenges make developing compelling personal data applications difficult. First, personal data is stored in isolated silos. Thus, relationships between data from different providers are missing, data must be queried by source of origin rather than meaning and the persistence of different types of data differ greatly. Second, interfaces to these data are inconsistent and complex. In turn, developers are forced to interleave SQL with Java boilerplate, resulting in error-prone code that does not generalize. Our solution is Epistenet: a toolkit that (1) unifies the storage and treatment of mobile personal data; (2) preserves relationships between disparate data; (3) allows for expressive queries based on the *meaning* of data rather than its *source of origin* (e.g., one can query for *all communications with John while at the park*); and, (4) provides a simple, native query interface to facilitate development.

ACM Classification Keywords
H5.m. Information interfaces and presentation (e.g., HCI): Miscellaneous.

INTRODUCTION

The promise of intelligent and personalized experiences motivates many of the most compelling smartphone application concepts. Intelligent assistants such as Siri, Google Now, and Cortana present their users with relevant information based on past behavior. Application launchers like Cover [23] and Aviate [24] predict what applications a user will use in different contexts to make them easier to launch. Logging applications like RescueTime [25] and Moves [26] let users collect and reflect on their own personal behavior. And, looking ahead, researchers envision other exciting personal data apps for domains such as health monitoring and ubiquitous smart environments.

MobileHCI '16, September 06 - 09, 2016, Florence, Italy
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-4408-1/16/09...$15.00
DOI: http://dx.doi.org/10.1145/2935334.2935349

An essential component for each of these applications is access to the rich archives of personal data stored by smartphones. Today, smartphones chronicle many of our everyday experiences ranging from our virtual interactions, such as our communication behavior and application usage, to our physical state, such as our location and physical activities [5,12]. As we delve further into the age of the Internet of Things, the breadth, fidelity and richness of the personal data available can only be expected to grow. However, at a time when distributing mobile applications is easier than ever before, several problems make developing personal data applications unnecessarily challenging.

First, personal data is stored in isolated silos. Accordingly, relationships between data from different providers are hidden—e.g., there is no link between the fact that a user was at the park and that she also played Angry Birds at 4:00pm. Consequently, while the most impressive personal data applications are those that make intelligent inferences on *interconnections* between data—e.g., predicting which emails a user would like to access based on who she just called or automatically logging one's activities during her visit to the park—accessing and making sense of these data interconnections is needlessly difficult.

Another problem is that personal data must be accessed by source-of-origin rather than meaning. This results in messy, un-generalizable data querying pipelines for aggregating semantically related data. For example, an application that tries to calculate relationship strength based on a user's communications with her contacts would not be able to simply query for the user's "communications"—it would have to separately query the call log, then the SMS inbox, and then any other third-party communication applications and interface with these data providers independently.

Yet another issue is the marked disparity in personal data persistence across different data providers. Consider, for example, the fact that call log data stretches back years yet GPS readings do not stretch back even seconds. While all of this data is enmeshed in a broader web of meaning, much of this meaning is lost simply because some data is sensed while other data is logged.

A final problem is programming complexity. Presently, interfacing with multiple personal data providers requires developers to interleave different programming patterns—a proven cause of programming errors [13,14]. For instance, on Android, to access personal *log* data (e.g., call logs and browser history), one needs knowledge of databases, SQL and Java iterator patterns to construct queries and process their results. In contrast, to access personal *sensor* data (e.g., accelerometer readings), one needs knowledge of sensor life cycles and event-based programming. Thus, the data acquisition and processing pipeline is cumbersome and error-prone even for experienced developers.

To address these issues, we propose Epistenet—a tool that semantically structures mobile personal data and offers a unified query interface to facilitate accessing semantically related personal data. Epistenet affords developers the ability to query for (1) data based on *meaning,* not source of origin; (2) data *related* to other, given data points, irrespective of data provider; and (3) histories of sensor data for a short period before and after other personal data points. Example queries include *all communications* (spans multiple data sources), *all application usage in the last day* (data that would not otherwise be accessible), or *web searches that occurred within 5 minutes of this phone call* (spans multiple data sources; relative to other data). In short, Epistenet *lowers the complexity* of accessing and processing related personal data, and *broadens the scope* of personal data that can be accessed with simple queries.

We offer the following novel contributions: (1) the design of Epistenet—including an ontology to organize personal data on mobile platforms in a knowledge graph, a native query interface that facilitates the retrieval of semantically related personal data, and data providers that afford developers histories of sensed information surrounding other personal data points; (2) an open-sourced proof-of-concept implementation of Epistenet for Android; (3) demonstrations of the utility and versatility of Epistenet through the creation of example applications; and, (4) a performance comparison of personal data querying tasks with and without Epistenet.

RELATED WORK

Epistenet is, in many ways, a deliberate refinement and extension of known ideas to a modern context. For example, the observation that data are more useful when relatable than when isolated is, alone, not novel. Harkening back to the Semantic Web, there has been a tradition of work aiming to break data out of their isolated siloes and into unified, semantically organized representations. The Resource Description Framework (RDF), for example, is a general metadata framework designed to be a knowledge representation mechanism for all web data [18]. Relatedly, SPARQL is a query language developed to query data encoded in RDF [27]. We note, of course, that there have been a number of efforts to port RDF and SPARQL clients to mobile platforms [4,7,16,22]. RDF and SPARQL alone, however, are not a solution to the problem of siloed data architectures: Rather, they *enable* potential solutions.

Yet, despite the continuing and formidable efforts of the semantic technologies community, we continue to see siloed data architectures on mobile platforms. This is not because designers of modern mobile architectures are ignorant of semantic technologies. Instead, we argue that semantic technologies have yet to see widespread use on mobile platforms because, alone, they do not provide a full solution that both: organizes personal data in a manner meaningful to the mobile personal data context *and* that facilitates the querying of this data in a manner that is simple for mainstream developers.

Epistenet builds on prior work in semantic technologies in that its goal is to organize data by meaning rather than source of origin. However, it differs in that it is architected specifically for the *purpose of integrating, organizing and facilitating access to personal data specifically in the mobile context*. Furthermore, it a full solution—rather than advocating the use of newly proposed standards and query languages, we provide an architecture that structures mobile personal data and exposes a native, unified interface to query and filter this data.

Epistenet is also related to prior work from other subfields. Indeed, prior work in sensor networks has tackled the related problem of consolidating information from multiple sensors. Researchers have proposed a number of solutions related to the storage [2,3], representation [3,10], and querying of data aggregated from raw sensor streams [1,17]. Sensor networks commonly represent data as schemaless tuple spaces [10], or collections of values concatenated as individual elements of a tuple [3]. These representations are terse, but can be difficult for application developers to work with. Generally, Epistenet varies from efforts in the sensor network literature in its scope (i.e., in semantically organizing *personal* data) and purpose (i.e., to simplify personal data application development).

Other work, e.g., Ohmage [21] and Lifestreams [9], has looked specifically at the problem of aggregating personal data, though these aggregators have typically focused directly on end-user use cases. For example, getting people to label personal data collections so that they can make sense of their personal data later, or constructing workspaces with relevant personal data. Epistenet differs from this work in that it is focused on making it easier for *developers* to create personal data applications that require access to *semantically related data*. In this way, we believe Epistenet is a tool that could be used *to create* systems like Ohmage and Lifestreams. Similarly, others have developed middleware solutions for efficient context extraction on smartphones [8,19]. Notably, Epistenet is *not a context extraction or inference application;* rather, it is a tool that should make data acquisition for these applications easier.

EPISTENET SYSTEM OVERVIEW

To reiterate, our goal with Epistenet is to *facilitate the retrieval and processing of semantically related personal data*, as these data afford developers the richest opportunities to create compelling apps. By personal data, we mean data about the user. So, Epistenet should index that the user sent an SMS to her friend, John, but not the phone's network connectivity when the message was sent.

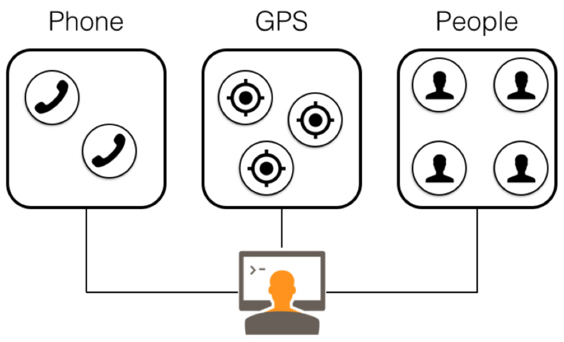


Figure 1: Without Epistenet, personal data is stored in silos. Developers must manually interface with each data provider, and relationships are hidden.

By facilitating access to semantically related data, we mean that Epistenet should make it easy to access both individual data points *and* the whole *web of interconnections between data points.* A personal data point is just one isolated datum about the user—for example, that the user sent an SMS message. But, that single data point is far less interesting than the subgraph of related data in which it is embedded—e.g., that the message was sent to Dad, while the user was at the park, and shortly after she called Mom. The isolated datum has little use alone, while the subgraph allows developers to create smart application launchers like Cover [23]. Presently, it is easy to get the datum but impossible to get the subgraph without manually constructing the links between data with every query.

With Epistenet, an individual data point should be an index into the larger graph of personal data, and queries should make it easy to retrieve subgraphs of related personal data.

Finally, by facilitating the processing of personal data, we mean that developers should be able to easily use the data they access. Presently, developers need to manually interleave SQL expressions with Java boilerplate to retrieve data, iterate through database records to process data, and understand database abstractions to modify data. Furthermore, they must do this separately for every data provider. Epistenet should provide native representations of personal data nodes, from any provider, that developers can process and manipulate as they would any other object.

To accomplish our goal, Epistenet (1) constructs and maintains a *knowledge graph* of smartphone personal data, and (2) provides a *unified query interface* that allows developers to query for subgraphs of the knowledge graph. Figures 1-2 illustrate how the knowledge graph changes the organization of mobile personal data.

Knowledge Graph

In the Epistenet knowledge graph, each personal data point is an *object*, contributed by a *data provider* and semantically organized into an *ontology.*

Objects

In Epistenet, objects are atomic units of personal data—for example, a phone call, a website visit, or a location ping.

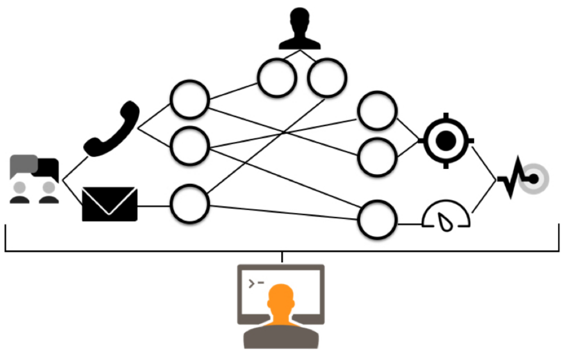


Figure 2: With Epistenet, personal data is stored and semantically structured in a knowledge graph. Relationships between data are exposed, and developers need only interface with one API.

Objects have three components: descriptive attributes, meta-attributes, and a link to at least one ontology class.

Descriptive attributes are key-value pairs of the core information encapsulated by an object. For example, a phone call might have four descriptive attributes: the communication partner(s), the duration of the call, the directionality of the call (outgoing or incoming), and whether or not it was answered. Meta attributes store information about the object. Each object has at least the four meta attributes—*timestamp:* when it was added; *persistence:* how long it should remain until being culled; *visibility:* the permissions required to see the object; and, *provider:* the provider that contributed the object.

Ontology class links serve two purposes. First, they "type" an object because an object must have the descriptive *and* meta attributes required by any ontology class to which it is linked. For example, objects linked to an ontology class describing website visits should have the website name and URL as descriptive attributes and the referring website as a meta attribute. Second, they organize objects into an information hierarchy—for example, objects about phone calls and SMS messages are both under "Communication".

A benefit of abstracting personal data points into "objects" is that they map nicely onto object-oriented programming objects. Consequently, with Epistenet, processing personal data points is the same as processing any other native object as opposed to the rows and columns of database records.

Ontology

The Epistenet ontology is a key enabler of much of its core functionality and is an adaptation of ideas from RDF applied to the mobile context. The ontology organizes objects so that data is structured by *meaning* rather than *source of origin*. It is composed of a set of *ontology classes* that are arranged hierarchically through *subsumption* (is-a) relationships and have links to objects—e.g., an accelerometer reading *is a* motion sensor reading.

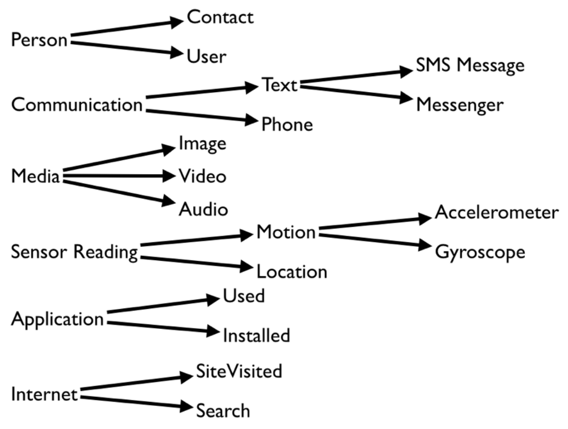


Figure 3: Default Epistenet ontology for smartphone personal data. Edges represent "is-a" relationships.

This semantic organization, when paired with Epistenet's unified query interface, affords the retrieval of entire subgraphs of related personal data. Indeed, we can access semantically related data by querying for objects linked to a shared top-level ontology class—e.g., we can find all data contributed by motion sensors, such as the accelerometer and the gyroscope, by querying Epistenet for objects with a link to the "MotionSensor" class and any class it subsumes.

An additional benefit of organizing data by meaning rather than source of origin is that developers need not care about how many data providers produce related data—they simply need to query for *what* data they want, not from *where* they want it. For example, consider the popular RescueTime app that logs how a user spends her time [25]. Without Epistenet, to count how much time a user spends browsing the web, a developer would need to independently query every web browser the user has installed on her phone—and this could vary widely across users. With Epistenet, data from all of those clients will be semantically structured under the same "SiteVisited" ontology class. Similarly, even if the user installs a new or deletes an old browser, the query to get browsing history need not change.

We constructed an initial, example ontology that organizes much of the personal data provided by default Android sensors and content providers (see Figure 3). To do so, we first enumerated all system-level (i.e., non-third-party) data providers found on modern Android phones. Then, we collaboratively and iteratively synthesized these data providers into hierarchical clusters [20]. While this ontology can be easily extended or replaced, it is sufficient for our proof-of-concept implementation of Epistenet.

Data Providers

Data providers produce personal data in the form of objects. Data providers can provide data in two ways: they can either *push* objects to Epistenet as new data are created, or Epistenet can *pull* data from them at regular intervals. Epistenet can aggregate information from arbitrarily many data providers—for example, system content providers such as SMS logs or hardware sensors such as the accelerometer.

Another contribution of Epistenet is that it offers **new data providers that store short histories of sensor and usage data**—e.g., a user's location for the past week, a 3-second window of motion data surrounding a phone call, and application usage logs. We wrote these providers to *bridge the disparity of data persistence* between sensor and log data. Both sensors and logs capture rich personal data, so there is no reason for call logs to stretch back years but for sensor data to not stretch back even seconds.

Consider the popular Moves app [26] that chronicles users' daily physical activities and location traces. Without Epistenet, developers need to manually store and associate location and motion sensor information. Furthermore, they would not be able to access this data prior to the user installing their application, and thus the application would initially offer no utility. With Epistenet, developers can access brief histories of location and related motion sensor information, even from *before* install-time, and would thus be able to immediately provide utility. These novel providers also eliminate the need for multiple applications to redundantly store histories of sensor information.

Graph Maintenance

Finally, to be mindful of the limited storage capacities of present-day mobile devices, the knowledge graph regularly culls personal data objects that have expired persistence fields. These persistence fields are initially set by contributing data providers, but can be overridden if the graph starts to overflow beyond a configurable, hard limit.

Unified Query Interface

The design and construction of the knowledge graph solves the problems of breaking personal data out of isolated siloes, as well as bridging the disparity of persistence between sensed and logged data. To address the remaining problems of reducing the programming complexity of accessing personal data and allowing query construction through *meaning* rather than *source-of-origin*, Epistenet provides a native unified query interface (UQI). The UQI is a single-point of entry that allows developers to *all* personal data stored in the knowledge graph, regardless of its source. Furthermore, the UQI allows developers to specify these queries with native code and also returns native objects.

Filters

Developers specify personal data queries with *filters*. Conceptually, filters are sieves that separate desirable objects from the rest of the knowledge graph. For example, an ontology-class filter could select objects that are linked to the "Person" class. Generally, filters can constrain any part of an object: its ontological links, descriptive attributes, meta-attributes or its relationship with other objects. To constrain one or many attributes of an object, a filter is given a set of *attribute constraints*—key-value pairs of attribute names mapped to an acceptable list of values.

Available attribute constraints include *greater/less than, equals, in set, out of set* and *regex matching*.

Filters can also be composited together to create more powerful relational queries. There are three filter composition operators in Epistenet that follow their standard definitions: *union* (return objects from both filters), *intersect* (return only objects in both filters), and *except* (return all objects in either, but not both, filters).

One of Epistenet's most powerful features is that filters can be created *relative* to any personal data point. In other words, *each object acts as an index into the larger knowledge graph*. These relative filters allow developers to expand from any given data point to a meaningful subgraph of the knowledge graph. For example, after retrieving a set of phone calls, it becomes trivial to access, for example, all locations a user visited 30 minutes before those phone calls.

While Epistenet filters may not be exciting in isolation, when used in tandem with Epistenet's knowledge graph, they make for a simple, powerful tool to surface complex relationships between personal data points. Figures 4-6 graphically depict how Epistenet's filters work.

Native APIs to Access, Process and Update Personal Data
Another feature of the UQI is that it is fully native. In our Android implementation, queries are specified in Java and return Java objects that represent their corresponding Epistenet objects. Accordingly, accessing, processing and updating personal data requires no knowledge of databases or query languages. This ensures that data accessing and processing pipelines are generalizable and robust (because they are not tied to particular database schemas and URIs) in addition to being simpler to use (as developers do not need to shift programming mental models nor do they need to coordinate between different interfaces).

Summary of Key Features
To summarize, Epistenet is an immediately usable full-stack solution that organizes, unifies and facilitates access to mobile personal data as well as bridges the disparity in treatment and persistence between sensed and logged personal data. To do so, Epistenet offers the following four key features: (1) Epistenet semantically organizes personal data in a knowledge graph; (2) Epistenet filters allow developers to access personal data by their *meaning* and *relationships* with other personal data rather than their source of origin; (3) Epistenet provides historical sensor and application usage information for a brief time window around other personal data points; and, (4) Epistenet provides a native unified query interface (UQI) for developers to retrieve, process and/or update any personal data in the knowledge graph.

USAGE SCENARIO
The following scenario demonstrates how Epistenet helps Android developers address the many challenges in creating personal data applications—specifically those that require accessing sets of *interrelated* personal data. This scenario describes the development of an autobiographical authenticator: A form of authentication based on asking people questions about their day-to-day experiences that has been of increasing interest to researchers in recent years [6,11]. Semantically interconnected personal data is core to autobiographical authentication.

The developer in this scenario is Cedric, a young man who has experience with Java and creating user-facing Android apps, but has little experience with databases. Cedric is building an autobiographical authenticator akin to the one proposed by Das et al. [6]. His application will ask end-users a number of multiple-choice questions, with each question corresponding to a fact in the user's everyday experiences captured by their smartphones. Example questions the application should ask include "Who did you SMS message on Sunday, May 22nd, at around 9:40pm?", "Where were you on Monday, May 23rd, at around 1pm?" and "What website did you visit on Tuesday, May 24th, at around 3pm?". Figure 7 shows a screenshot of the final app.

Every question asked consists of three components: a *personal data point* that contains an answer to a question, a *set of answer choices* from which the user must select the correct answer, and a *set of contextual hints* that gives users hints of other things that happened around the same time as the *personal data point* in order to cue their recall of the correct answer. Each component requires Cedric to write code that dynamically aggregates, filters and processes data from a multitude of data providers, not all of which are natively present on Android (e.g., sensor history providers).

In implementing a question generator, Cedric creates an

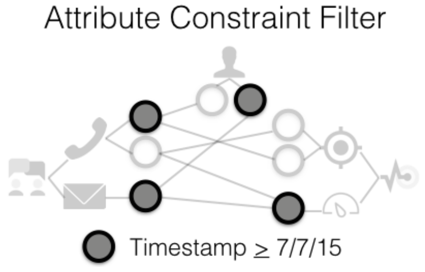


Figure 4. Attribute constraint filters select object that have descriptive or meta-attributes that meet specifications.

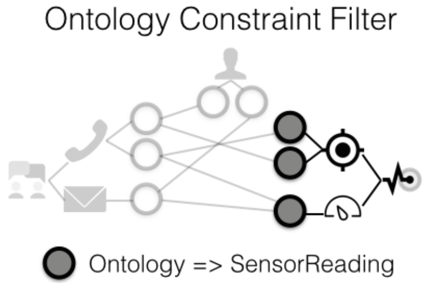


Figure 5. Ontology constraint filters select object that are linked to certain ontology classes or any subclasses.

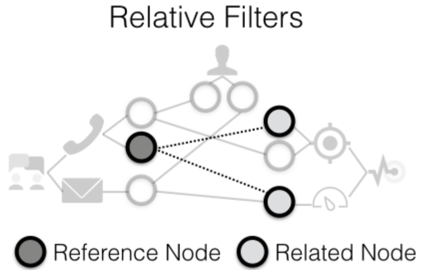


Figure 6. Relative filters select objects that have a certain relationship with a reference node.

abstract Question superclass that provides a basic structure for each question he would like his application to support:

```
abstract class Question {
  abstract void findPersonalDataPoint()
  abstract String[] getAnswerChoices();
  abstract String[] getContextHints();
}
```

The methods of this abstract class correspond to the aforementioned required components for each question. Cedric's task is to implement subclasses for each of the eight question types he wants his application to support. This process differs markedly with and without Epistenet. To simplify the scenario, we take the question "What website did you visit on {{time}}?" as an example. To implement support for this question, Cedric creates the class `BrowserQuestion extends Question`. The following sections describe how Cedric will implement the methods in `BrowserQuestion` both with and without Epistenet.

findPersonalDataPoint()

To implement the `findPersonalDataPoint()` method, Cedric needs to find a *personal data point* that represents a valid answer to a given question. For the `BrowserQuestion`, an appropriate personal data point would be a record of any website that Cedric visited. The name of the website in this data point would be the answer to the question. Cedric also needs to restrict the matching personal data point to one that occurred between 3 and 24 hours ago so that users can reasonably be expected to remember the answer but also so that a shoulder surfer would not be able to guess the answer based on a brief, momentary observation.

Using Epistenet

Cedric needs an `EpistenetObject` linked to the "`SiteVisited`" ontology class with a timestamp within the past 3 to 24 hours. He finds a personal data point that meets all of these criteria with the `getRandom` method of the `Epistenet` UQI, which takes a `Filter` object as an argument and randomly selects one of the matching `EpistenetObjects` to return. The `Filter` object Cedric passes in is an *intersect composite filter* that intersects the results of an *ontology class filter* that constrains objects to those that have a link to the `SiteVisited` ontology class, and a *meta-attribute filter* that constrains objects to those that have a timestamp between 3 (`highTime`) and 24 (`lowTime`) hours ago. Thus, the entire process of finding an appropriate personal data point is simply:

```
EpistenetObject fact = Epistenet.getRandom(
  Filter.constructOntologyClassFilter(
    OntologyClass.SiteVisited).intersectWith(
      Filter.constructTimeRangeFilter(
         lowTime, highTime));
```

Without Epistenet

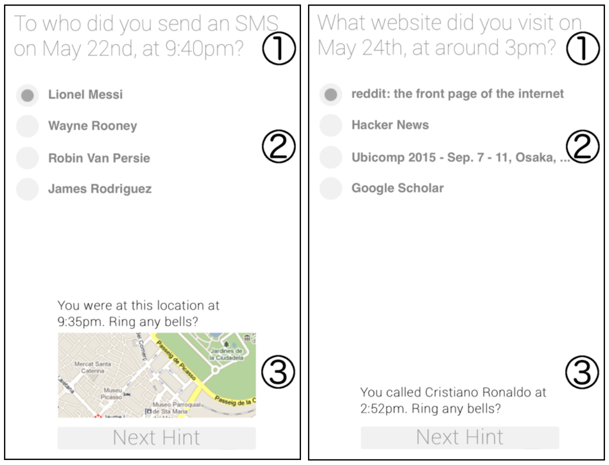
Without Epistenet, in order to access data, Cedric needs to understand the specifics of how and where data is stored. This is where Cedric's limited exposure to query languages and databases becomes a hindrance. Indeed, to find a personal data point to initialize the question, Cedric must use Android's default content provider API [28] to access the user's browser history by calling the `query` method of the `ContentResolver` class. To access one random website that the user visited between 3 (`highTime`) and 24 (`lowTime`) hours ago with the standard content provider APIs, Cedric writes the following:

```
Cursor c = mContext.getContentResolver().query(
  Browser.BOOKMARKS_URI,
  new String[] {
    Browser.BookmarkColumns.TITLE,
    Browser.BookmarkColumns.DATE
  },
  Browser.BookmarkColumns.DATE + " >= ? AND " +
  Browser.BookmarkColumns.DATE + " < ?",
  new String[] { lowTime, highTime},
  “RANDOM() LIMIT 1”);
```

The first argument to the query method is the "URI" (location) for the content provider. This requires a change in mental model: Cedric is no longer dealing with simple Java code, but with a RESTful resource request—a paradigm quite distinct from the rest of his application. This mental model shift is likely to make him less efficient and introduce more bugs in his code [13,14].

Next, Cedric passes in a `String` array of the database "columns" he wants returned. In this case, he wants access to the website title (which will ultimately be the answer to the question) as well as the date (which will ultimately be needed to fill in the question text, as well as to find distractor answers and contextual hints). Then, for the third and fourth arguments, Cedric must manually write and pass in a SQL *where clause* in order to restrict the return values to only those websites the user browsed in the last 3 to 24 hours. Finally, in order to select one random record out of all the matching records, Cedric must write a SQL *order by* and *limit* statement. In this case, he must order by the SQL `RANDOM()` helper method to shuffle the records and then `LIMIT` the query results to just the first.

The result of this entire query is a Cursor object that iterates


To who did you send an SMS on May 22nd, at 9:40pm?

Lionel Messi
Wayne Rooney
Robin Van Persie
James Rodriguez

You were at this location at 9:35pm. Ring any bells?

What website did you visit on May 24th, at around 3pm?

reddit: the front page of the internet
Hacker News
Ubicomp 2015 - Sep. 7 - 11, Osaka, ...
Google Scholar

You called Cristiano Ronaldo at 2:52pm. Ring any bells?

Figure 7: Example autobiographical authentication questions. There are three components: (1) The question asked. (2) The answers. (3) The hints.

over the rows and columns of matching database records. Now, in order to extract the site title of the website as well as time the user visited the website, Cedric must write the following meticulous code:

```
String answerTitle; long answerTime;
if (c != null) {
  if (c.getCount() > 0) {
    c.moveToFirst();
    answerTitle = c.getString(c.getColumnIndex(
      Browser.BookmarkColumns.TITLE));
    answerTime = c.getLong(c.getColumnIndex(
      Browser.BookmarkColumns.DATE));
  }
  c.close();
}
```

This interleaving of SQL with Java boilerplate is messy and requires Cedric to constantly switch between programming mental models. Despite having limited exposure to databases and query languages, Cedric must develop a relatively deep understanding of database abstractions and query languages to get something as simple as a website visit. Furthermore, all of this code gets him *only* the web history from the default browser on the user's phone. If the user has multiple web browsers, Cedric will have to (1) enumerate all web browser history providers and (2) repeat the process of querying for and extracting information from each of these providers. Additionally, as each of these providers will be in a different location and are likely to have different schemas, relatively little of the code he has written for the default web provider will generalize.

getAnswerChoices()

Next Cedric must implement `getAnswerChoices()`, which should return an array including the correct answer and a set of plausible but incorrect *distractor answers*. Distractor answers should be ontologically similar to the "correct" answer. So, if the "correct answer" is the name of a website the user visited, distractor answers should also be the name of websites the user might visit. Additionally, distractor answers should not be reasonably confused with the correct answer. For our running example, a bad distractor answer would be the name of a website a user visited only five minutes prior to when she visited the "correct" website. Thus, Cedric needs to write code that finds semantically similar "facts" that are far away, in time, from the original.

Using Epistenet

Cedric uses the `Epistenet.getAttributeOf` static method of the UQI that takes in a `Filter` object and descriptive attribute name, and returns the corresponding descriptive attribute of *all* `EpistenetObjects` that match the `Filter`. He uses the method `Filter.constructRelativeTo` method to create a `RelativeFilter` that adds filter constraints *relative* to a reference `EpistenetObject` (in this case, the original fact). To specify the relationship that distractor answers should have to the correct answer, Cedric uses two builder methods of the `Filter` class: `lessThanTimestampWithOffset()` (to only match `EpistenetObjects` that occurred at least 6 hours prior) and `sameOntologyClass()` (to only match `EpistenetObjects` from the same ontology class). All `Filter` builder methods return a reference to the modified `Filter` object, so Cedric concisely chains these two calls. The resulting code is:

```
String[] distAns = Epistenet.getAttributeOf(
    Filter.constructRelativeTo(fact)
     .lessThanTimestampWithOffset(6*60*60*1000)
     .sameOntologyClass(), "website-title");
```

Without Epistenet

The full implementation of this method without Epistenet is too long to fully present. Indeed, for every web browser history provider, Cedric has to write code similar to his query in `findPersonalDataPoint()`, except the *where* clause would have to written in relation to the `answerTime` variable he extracted before and the "`LIMIT 1`" portion of the *order by* clause should be omitted so that he can get multiple matching records. Then, for *each* query response, he would have manually iterate through rows as well as columns in order to extract the relevant information (the website title) and store it an a `String` array.

```
String[] distAns = new String[c.getCount()];
int counter = 0;
if (c != null) {
  if (c.getCount() > 0) {
    c.moveToFirst();
    while (!c.isAfterLast()) {
      distAns[counter++] = c.getString(c.getColumnIndex(
          Browser.BookmarkColumns.TITLE));
      c.moveToNext();
    }
  }
  c.close();
}
```

This code is redundant and error-prone, and again requires Cedric to shift programming mental models. Moreover, this code does **not** generalize across web browsers because each browser can have a different database schema.

getContextHints()

Finally, Cedric must provide users with a set of contextual hints to assist users if they have trouble answering the question. For example, knowing that one was at the park when browsing a website might trigger the memory of something that the user read and, in turn, the name of the site. Cedric decides that a good hint is anything that the user did 10 minutes before or after visiting the "correct" website.

Using Epistenet

Relative queries are again perfectly suited to the task. Cedric uses the object representing the correct answer, stored in the `fact` variable, as the reference object. He calls the `withinTimeRange` method of the `RelativeFilter` class and passes in the value of 10 minutes in milliseconds, indicating that he wants any fact that occurred in the 10 minute time period before or after the reference object.

```
EpistenetObject[] hints = Epistenet.get(
  Filter.constructRelativeTo(fact)
        .withinTimeRange(10*60*1000));
```

Cedric now has *all* personal data captured in the 10 minutes surrounding the correct answer, regardless of the number and type of personal data providers on the user's phone. He can use the same post-processing techniques he earlier employed to render the hints to his users.

Without Epistenet

Again, the full implementation of this method without Epistenet is far too long to fully present. Cedric must compose schema-specific queries *for each of the dozens of personal data providers* that could provide a hint in the corresponding time period. Each of these queries on the surface looks similar to the `findFactAndInitialize()` method, but the specific content provider URIs, column names, and constraints are specific to the provider. Post-processing will again involve a cumbersome, error-prone conversion of provider-specific table columns into a "hint" `String` that will then be aggregated across all hint providers. Despite little experience with databases, Cedric must dive deep into this world in order to access the personal data he needs.

Summary of Differences

To summarize, generating autobiographical authentication questions requires accessing personal data that *spans many data providers* and that *relates to other personal data* in very specific ways (i.e., to create distractor answers and get contextual hints). With Epistenet, the code required to access the required this data is simple and concise because: (1) Epistenet allows developers to query for personal data based on *meaning;* (2) each personal data point is an index into the larger knowledge graph; and, (3) Epistenet provides a native query interface that developers can use without switching programming mental models. But, without Epistenet, doing the same requires hundreds of lines of meticulous, error-prone code that is hard to generalize.

IMPLEMENTATION

We implemented a proof-of-concept version of Epistenet on Android 4.0.4[1]. Our implementation comprised of two components. The first is an Android service that indexes and organizes data in an ontology, as well as exposes an interface to access the data through a standard Android content provider. The second is an external JAR—EpistenetJAR—that serves as a "query interface" that developers can include in their Android applications to query Epistenet on devices that have the Epistenet application installed. EpistenetJAR itself is a lightweight library and can be included into any Android application. Note, however, that this is only a proof-of-concept implementation of Epistenet to demonstrate its utility, so specific implementation details are not critically important. Ideally, Epistenet would be integrated into the OS so that a separate application would not be necessary.

To construct the knowledge graph, we indexed many system-level content providers and sensors. Specifically, we indexed the content providers for the SMS inbox; the call log; audio, video, and image media; web browser search and visit history; and calendar. In addition, we created content providers for the location and accelerometer sensors, as well as for application use and installations. These data providers were polled independently at regular intervals through a background service. Motion sensor data was retained if it occurred within a 3-second time window of another data point. To support push notifications for indexing data as they were created, we also created `BroadcastReceiver` classes for new SMS messages, calls, pictures, videos, and installed packages. A background service also culled expired objects.

To avoid undermining Android permissions, data indexed by Epistenet had its visibility meta-attribute set to the default permissions required to access that data on Android. For example, all objects contributed by the GPS sensor have a visibility value of "ACCESS FINE LOCATION"—the permission required by Android to access the GPS sensor. In this way, Epistenet only returns data for which a requesting application has permission to access. Notably, we are **not** arguing that this is sufficient to guarantee that no new privacy holes are created, only that we have taken steps to mitigate the introduction of these holes. We discuss privacy implications more thoroughly later.

OTHER DEMONSTRATIVE EXAMPLES

Personal Data Tracker & Information Management

Personal data trackers, like RescueTime [25] and Saga [29], require access to collections of related personal data to provide utility. Representative of these types of applications, we created a lifelogging application that shows users a chronological view of their day-to-day lives—for example, where they were, who they contacted, what applications they used and what websites they visited.

With Epistenet's UQI, aggregating and filtering data spanning many data providers is simple. We displayed a list of dates that users could tap to get information about their activities on that date. For any date that a user clicked on, we simply queried Epistenet with a meta-constraint filter that constrained the timestamp of matching Epistenet objects to the selected 24-hour period.

```
long millisDay = 24*60*60*1000;
Epistenet.get(Filter.constructTimeRangeFilter(
  now - daysAgo*millisDay,
  now - (daysAgo - 1)*millisDay));
```

Notably, *the code remains the same irrespective of the number personal data providers on the phone*. Furthermore, without writing any complex code to "listen" to location pings and application usage events, we are *able to show users where they were for the past few weeks*. To make the application interactive, we allowed users to click on any rendered fact to find other similar facts that occurred in the past two weeks. For example, if the user clicked on a phone call with her contact John, she would see every other phone call she had with John in the previous 2 weeks. With *relative queries*, implementing this functionality was as simple as the following snippet:

[1] Open sourced at: https://github.com/scyrusk/epistenet.git

```
Epistenet.get(Filter.constructRelativeTo(choice)
  .sameOntologyClass()
  .sameDescriptiveAttribute("contact")
  .withinTimeRange(2*7*24*60*60*1000));
```

Closeness Ordered Contact List

Another common use for personal data is to customize application experiences. For example, Cover [23] uses personal data to predict what applications a user is likely to use and reorders those predictions to make them easier to launch. Accordingly, we made an app that reorders contacts based on how frequently they communicate with the user. The challenge here is that communication comes in many forms—for example, through e-mail, social media, or phone calls. With Epistenet's ontology, it is easy to access *all* communications and group them by a common attribute.

```
Map<String, EpistenetObject[]> comms =
  Epistenet.getAndGroupResultsBy(
    Filter.constructOntologyClassFilter(
      OntologyClass.Communication),
    "partner");
```

The comms variable now contains a Map with *keys* pertaining to the "partner" descriptive attribute, and *values* containing all objects linked to the "Communication" ontology class with the partner descriptive attribute equal to the corresponding *key*. So, to present a user with a frequency-ordered contact list, we need only rearrange the keys (i.e., names of contacts) of the comms Map to be in descending order of the length of its values (i.e., an array of EpistenetObjects representing communications with a given contact), and then show the ordered *keys* to the user.

If This Then That

Applications that use personal data conditions to "trigger" functionality are also common. For example, applications like If This Then That [30] allow end-users to specify "recipes" of what they would like to happen if certain conditions are met—for example, if a user contacts someone five times in a week, then "favorite" that contact.

Recipe triggers, which are just union or intersection of many different conditions, nicely map to Epistenet's filter compositions. For example, consider the recipe: if the user has not been back home in the past week, then notify the user that he should call home. Checking if the trigger has been met requires a simple composition of three filters:

```
boolean wasHome = !Epistenet.any(
  Filter.constructOntologyClassFilter(
    OntologyClass.LocationReading)
  .intersectWith(
    Filter.constructAttributeEqualsFilter(
      "location-name", "home")
  .intersectWith(
     Filter.constructTimeRangeFilter(
      now - 7*24*60*60*1000, now));
```

If no matching objects were found, then the Epistenet.any method will return false and wasHome will be set to true.

DISCUSSION & POTENTIAL LIMITATIONS

Query Performance. One concern is whether Epistenet has acceptable query performance. To evaluate query performance, we enumerated 11 data querying tasks required by both the autobiographical authentication demo (e.g., to find a matching personal data point for the question "Who did you SMS at {{time}}?") and the personal data tracker demo (e.g., everything the user did in a 24 hour period). We implemented both an Epistenet and non-Epistenet version of these tasks. All of these tasks required querying multiple data sources.

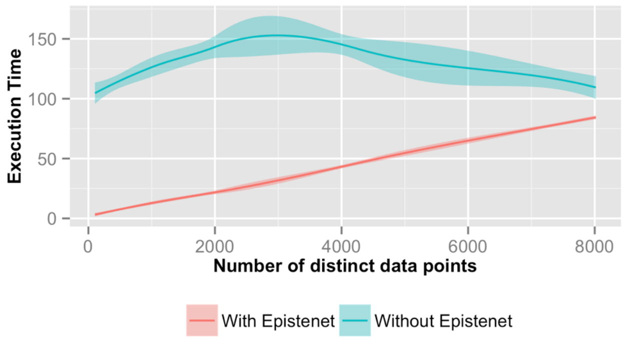


Figure 8: Execution time (in milliseconds) comparison across all tasks for both Epistenet and the default android content providers, with 95% confidence intervals.

We simulated each querying task across six different values of the total number of data points available: 100, 500, 1000, 2000, 4000, and 8000 data points. We ran each task 100 times at each value of total data points for both the Epistenet and non-Epistenet implementations. Figure 8 shows the results. As is expected, Epistenet is often faster than the default Android content providers, though its runtime, expectedly, does linearly increase with the number of indexed objects. Perhaps most telling, though, is that all queries, both Epistenet and non-Epistenet, took fewer than 200 milliseconds to run. Thus, *query performance does not appear to be a large concern for Epistenet*. We also believe that implementation changes can improve performance.

Battery Consumption. Epistenet takes a small toll on battery life, however exactly how much should vary markedly across hardware, a user's activity level, and the existing applications on a phone [31]. Anecdotally, on a test phone (Samsung Galaxy S3) running Epistenet for several weeks, Epistenet did not drain battery life to a point where the phone needed to be charged more than once per day.

Privacy. While we have taken steps to avoid undermining Android permissions, our approach does not address all privacy issues. For example, Android does not presently have a permission affording applications a recent history of location readings—only from the point of installation. In its ideal form, Epistenet's host system would have permissions for all data accessible through it. But, there is a broader privacy issue: By making it easier for developers to create more powerful personal data apps, we also make it easier for developers to violate end-user privacy. As prior work points out [15], this is a conundrum in ubiquitous computing—the same sensors and logs that allow us to create data rich, context-aware, personal data applications can also be used to create a privacy-invasive surveillance infrastructure. We acknowledge that this is a sensitive problem, but believe it to be broader than Epistenet.

CONCLUSION

Presently, developing compelling personal data applications on mobile platforms is difficult because: personal data is stored in isolated silos, thus suppressing relationships between data from different providers; interfaces to these data are inconsistent and force developers to query for data based on source-of-origin rather than meaning; and, developers are forced to interleave multiple programming abstractions to construct even simple queries, resulting in error-prone, un-generalizable data processing pipelines. To address these challenges, we implemented and evaluated Epistenet: an architecture that facilitates the programmatic access and processing of semantically related personal data. In short, Epistenet constructs and maintains a knowledge graph that structures mobile personal data by meaning and provides a native, unified query interface to access subgraphs of this data. We argue that this represents a promising first step in creating a solution that facilitates the development of intelligent smartphone applications built on rich, semantically interrelated personal data.

ACKNOWLEDGEMENTS

We thank Michael Villena, Barath Chandrashekar, and our reviewers for their helpful feedback.

REFERENCES

1. P. Bonnet, J. Gehrke, and P. Seshadri. 2000. Querying the physical world. *IEEE Personal Communications* 7, 5: 10–15. http://doi.org/10.1109/98.878531
2. Philippe Bonnet, Johannes Gehrke, and Praveen Seshadri. 2001. Towards Sensor Database Systems. *Proc. MDM'01*, 3–14.
3. Carlo Curino, Matteo Giani, Marco Giorgetta, Alessandro Giusti, Amy L Murphy, and Gian Pietro Picco. 2005. Mobile Data Collection in Sensor Networks : The TinyLime Middleware. *Elsevier Pervasive and Mobile Computing Journal* 4: 446–469.
4. Mathieu d'Aquin, Andriy Nikolov, and Enrico Motta. 2011. Building SPARQL-Enabled Applications with Android devices. *Proc. ISWC'11*.
5. Sauvik Das, LaToya Green, Beatrice Perez, Michael Murphy, and Adrian Perrig. 2010. Detecting User Activities Using the Accelerometer on Android Smartphones. TRUST-REU Tech Reports
6. Sauvik Das, Eiji Hayashi, and Jason Hong. 2013. Exploring Capturable Everyday Memory for Autobiographical Authentication. *Proc. UbiComp'13*.
7. Jerome David and Jerome Euzenat. 2010. Linked data from your pocket . *Proc. ISWC'10 Demo Track*.
8. Anind K. Dey. 2001. Understanding and Using Context. *Personal and Ubiq. Computing* 5, 1: 4–7.
9. Eric Freeman and David Gelernter. 1996. Lifestreams. *ACM SIGMOD Record* 25, 1: 80–86.
10. David Gelernter. 1985. Generative communication in Linda. *ACM TOPLAS* 7, 1: 80–112.
11. Alina Hang, Alexander De Luca, and Heinrich Hussman. 2015. I Know What You Did Last Week! Do You? Dynamic Security Questions for Fallback Authentication on Smartphones. *Proc. CHI'15*.
12. A. M. Khan, Y.-K. Lee, S. Y. Lee, and T.-S. Kim. 2010. Human Activity Recognition via an Accelerometer-Enabled-Smartphone Using Kernel Discriminant Analysis. *Proc. FIT'10*, 1–6.
13. Andy J. Ko, Brad A. Myers, and Htet Htet Aung. 2004. Six Learning Barriers in End-User Programming Systems. *Proc. VLHCC'04*, 199–206.
14. Andy J. Ko and Brad A. Myers. 2003. Development and evaluation of a model of programming errors. *Proc. HCC'03*, 7–14.
15. Marc Langheinrich. 2001. Privacy by Design - Principles of Privacy-Aware Ubiquitous Systems. *Proc. Ubicomp'01*, 273–291.
16. Danh Le-Phuoc, Josiane Xavier Parreira, Vinny Reynolds, and Manfred Hauswirth. 2010. RDF on the go: An RDF storage and query processor for mobile devices. *Proc. CEUR Workshops* 658: 149–152.
17. Samuel Madden, Michael J Franklin, Joseph M Hellerstein, and Wei Hong. 2003. The design of an acquisitional query processor for sensor networks. *Proc. SIGMOD '03*, ACM Press, 491–502.
18. E Miller. 1998. An Introduction to the Resource Description Framework. *D-Lib Magazine*.
19. Suman Nath. 2012. ACE. *Proc. MobiSys'12*, ACM Press, 29–42.
20. Natalya F Noy and Deborah L Mcguinness. 2000. Ontology Development 101 : A Guide to Creating Your First Ontology. *SKSL Technical Report KSL-01-05*, 1–25.
21. H. Tangmunarunkit, J. Kang, Z. Khalapyan, et al. 2015. Ohmage. *ACM Transactions on Intelligent Systems and Technology* 6, 3: 1–21.
22. Roberto Yus, Carlos Bobed, Guillermo Esteban, and Fernando Bobillo. 2013. *Android goes Semantic : DL Reasoners on Smartphones*.
23. Cover Lock Screen. Retrieved from https://play.google.com/store/apps/details?id=com.coverscreen.cover&hl=en
24. Yahoo Aviate Launcher. Retrieved from https://play.google.com/store/apps/details?id=com.tul.aviate&hl=en
25. RescueTime. Retrieved from https://rescuetime.com
26. Moves App. Retrieved from https://moves-app.com
27. SPARQL. Retrieved from http://www.w3.org/2009/sparql/wiki/Main_Page
28. Content Provider. Retrieved from http://developer.android.com/guide/topics/providers/content-providers.html
29. SAGA. Retrieved from http://www.getsaga.com
30. If This Then That. Retrieved from https://ifttt.com
31. Power Profile for Android. Retrieved from https://source.android.com/devices/tech/power.html