Small Data: Applications and Architecture

Cheng-Kang Hsieh*, Faisal Alquaddoomi†, Fabian Okeke‡, John P. Pollak§, Lucky Gunasekara¶ and Deborah Estrin‖
* UCLA CSD; Los Angeles, CA, USA (changun@cs.ucla.edu)
† UCLA CSD; Los Angeles, CA, USA (faisal@cs.ucla.edu)
‡ Cornell CSD; Ithaca, NY, USA (fno2@cornell.edu)
§ Cornell Tech; New York, NY, USA (jpp9@cornell.edu)
¶ Cornell Tech; New York, NY, USA (llg24@cornell.edu)
‖ Cornell Tech; New York, NY, USA (destrin@cornell.edu)

***Abstract*—Small data are the digital traces that individuals generate as a byproduct of their daily activities, such as: communicating through email or text; buying groceries or ordering delivery; or going to work on foot or by car. These traces can empower individuals to gain insights into their behavior, personalize their care, improve their relationships, motivate achievement of goals, and broadly improve their quality of life. As such small data are both byproducts of today's and drivers of tomorrow's ubiquitous computing applications. The contributions of this paper are twofold: we motivate the requirements for a small data ecosystem and supporting architecture, and present a critical component – Lifestreams Database (DB) – which is evaluated using three exemplar apps. Lifestreams DB extracts, processes, and models diverse traces from data silos and enables various small data applications through simple SPARQL queries. Its soft-state design provides storage-efficiency, robustness, and query performance for processing small data.**

Keywords–small data; linked data; knowledge representation.

I. INTRODUCTION

Small data are "digital traces", records of our activities that are stored as we interact with the world around us. These traces are passively produced when we use tools and services that maintain logs: credit cards, grocery receipts, websites and other streaming content services, browsers themselves, etc. They can also be intentionally produced and tracked by wearable sensors, including mobile phone applications. It is well-known that service providers derive value from this information – usage metrics and demographic information, all personal data, are routinely employed to help direct advertisement and optimize products. We argue that this data can and should provide value for the producers of this data as well. As a natural extension of prior ubiquitous computing applications, **small data apps** will emerge as an important class of ubicomp applications that concern themselves with deriving insight from personal data at the user's request and with their oversight.

For example, a small data app may promote healthier eating by coaching users to take the planning actions needed to prepare meals at home. The app would utilize grocery and online food delivery history, browser history, and Moves or Foursquare data to build a model of meal preferences. The user could then receive prompts at their desired frequency about which recipes they are likely to enjoy, and suggestions for additions to their grocery shopping list to enable them to prepare these meals at home. The app could incentivize this with informative comparisons of calorie and cost savings, or could be tied to more intentional gamification. Another small data app could allow independent living elderly to share how they are doing without sharing every detail of what they are doing. The app would make use of passively collected small data streams such as email, activities, and mobile phone usage to create a personalized model of the user's activity, well-being, and degree of social engagement. Rather than exposing the model itself, the app would expose deviations from the model to make family and friends aware of changes to a person's state without divulging detailed information. Such an app can support many types of relationships, including family and friends separated geographically, or other support-network relationships such as social workers, caregivers, and coaches. We describe these concepts in greater detail in section III.

The central role of a small data architecture is to facilitate application-level access to a person's diverse information sources on their behalf. While individual service providers, such as Google, Facebook, and Amazon each have information about many aspects of our behavior, they are limited in how specifically they personalize by the terms of their end-user licensing agreements and a need to preserve users' trust. They also do not each have access to all data of interest. Because of this, there is an opportunity in the market for providers to give users access to their individual data in various forms (application programming interfaces, downloads, email receipts), and for third-party products to emerge that integrate with that user's data in the same way that third party mobile apps make use of mobile-device data. These third party apps would serve the end user without degrading the large-service provider's position, and in fact have the potential to solidify the user's sense of the service provider's utility and trustworthiness. Note that we are promoting that users be given access to their data and not making any statement about data ownership. We are also not addressing the very important policy question regarding service providers making user data available to third parties directly.

As mentioned, service providers have difficulty providing apps that cut across multiple data sources or mine too deeply into their users' data. In contrast, a small data app leverages the user as the common denominator, and can take advantage of the trend for service providers to support application programming interfaces (APIs) for individuals to their data. The user has both the access and authority to collect and aggregate data across these providers, allowing for powerful and comprehensive insights that, by virtue of the fact that they are initiated and consumed by that same user, can be much more focused in their oversight and suggestions. We anticipate and favor broad provision and adoption of systematic programmatic access to personal data for the end users. However, the need for a small-data application architecture need not wait for, nor will it be obviated by, future developments. Already, today, users can

 ISBN: 978-1-61208-631-6

obtain access to their data, albeit through idiosyncratic and sometimes ad-hoc channels: e-receipts, diverse APIs, browser plug-ins, etc. Even with access to these data, infrastructure is still required to process these traces into formats that are useful and actionable to the individual. Since most individual users do not develop their own software, we are targeting support for small-data app developers who will implement apps on the behalf of this growing user base; just as they have driven the development of third party apps for smartphones [1]. This approach is aligned with the emerging Social Web activities in W3C [2].

Our vision is to create a **small data ecosystem** in which small data apps can be readily developed and deployed atop an infrastructure that standardizes their inter-operation and addresses concerns that are common across apps, such as helping to ensure security and reducing redundancy in storage and computational resources, as well as resolving policy/legal questions that are outside the scope of this paper. The vision is, again, driven by the individual as the common denominator, and rightful beneficiary, of access to their data.

We describe the core components of a small data architecture using three exemplar applications, and present a specific system-design for the most central of these components – Lifestreams Database (hereafter "Lifestreams DB"). Lifestreams DB is designed to extract and process diverse digital traces from various sources and make them available to the client applications for further analysis or visualization. **Data interoperability** is an important requirement for such a system as it allows one to gain insights from the combination of data that were originally locked in their own data silos. Lifestreams DB extracts raw data from these data silos, and transforms them into a standardized Resource Description Format (RDF) that allows one to join these digital traces against each other and with external RDF data sources (e.g., fuse nutrition information with users' online shopping records.)

Unlike many enterprise settings, small data differs in the fact that most of original sources (e.g., Google, Facebook, etc.) persist users' data in their own databases and individually provide security and access control. Therefore, it may be wasteful, or even harmful to the users' security and privacy for Lifestreams DB to permanently replicate these data in one place. Motivated by this distinction, we propose a **soft-state design** that, while providing client applications with virtual access to all the data, only caches a part of it locally, and reproduces the rest on demand. Such a design introduces two important advantages in the context of small data. First, our soft-state model discourages our system from becoming a data "honeypot" that attracts attacks from malicious entities since only a limited amount of information is cached in the system at any given time. Second, it requires much less storage and allows the system to scale to serve a large number of users or integrate with more diverse information beyond its storage capacity. We also provide an encryption mechanism that encrypts the sensitive data to further protect the user.

After introducing related work in section II, we present three small data applications in III and use them to identify cross cutting application requirements. We provide a brief overview of our architecture in section IV, then go into depth on the main contribution of this work, Lifestreams Database (DB), in section V. Section VI contains the results of performance analyses for simulated workloads on a sample of simple and complex query types. Finally, section VII provides some observations and outlines future work.

II. Related Work

Small data are fueling a new genre of personalization technologies. Recommender systems have been some of the most successful applications in this domain to date as evidenced by recommendations for music in Pandora, consumer goods in Amazon [3], articles in Wikipedia [4], and locations in Foursquare [5]. These systems rely heavily on the users' application-specific histories, such as queries, clicks, ratings, and browsing data that result from interacting with their product. Small data can enable far more immersive recommender systems that take into account a larger space of user needs and constraints. In particular, they can benefit from user models derived from both more diverse and longitudinal data (e.g., features and dynamic patterns in: daily travel patterns, consumption from gaming to dining, interests and sentiment expressed in personal communication, etc.). General-purpose recommendation frameworks such as MyMediaLite [6] and LensKit [7] (to name a few) could make use of small data to learn these kinds of broad user models, but they require a front-end component to fetch user's data and drive the framework with appropriately-formatted inputs.

Small data's goal of providing individuals with transformative insights into their behavior is aligned with that of the Quantified Self (QS) movement [8]. In QS studies, individual experimenters engage in the self-tracking of biological or behavioral information using commercial devices such as Fitbit and myZero sleep trackers, or personal testing services such as 23AndMe, and many systems have been developed to help integrate and visualize QS data [9]. Even prior to QS's popularity, research projects such as Ubifit and BeWell demonstrated the potential of making personal data actionable [10][11]. More recent work, i.e., EmotionCheck [12], has demonstrated that not only QS data itself, but a user's trust in the tool, can serve as effective leverage for behavioral change. Small data, however, differs from earlier studies in its focus on harnessing data that are (a) generated as byproducts of interacting with services and (b) that are readily available, versus having to be manually collected or otherwise procured. These data can be complementary to or serve as a proxy for some of the data that QS studies collect.

Small data are also related to Personal Information Management (PIM) systems [13]. This line of work covers a broad range of environments from desktops [14][15], to connected-devices in the home [16][17], to e-learning [18] and health information management systems [19]-[22]. Our work is complementary to these systems' focus on information organization and retrieval, by providing support for third party applications that would generate additional inputs to these systems through the processing of small data streams that are not yet accessible.

Small data shares similar data input with Personal Automation Engines. For example, Atomate [23] is a system that integrates individuals' social and life-tracking feeds into a unified RDF database, and automatically carries out simple tasks (e.g., messaging) when the incoming feeds satisfy user-defined rules. The service "If-This-Then-That" (IFTTT) [24], expanding on the same idea, compiles a large set of feeds that monitor various online and offline activities and can trigger a wide set of actions when a user-defined condition on a feed is satisfied. On a more application-focused and user-local

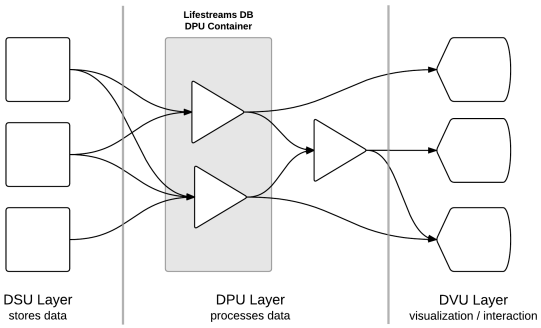


Figure 1. Small Data Architecture: illustrates the flow of data between Data Storage Units (DSUs), Data Processing Units (DPUs), and Data Visualizations Units (DVUs, e.g., apps).

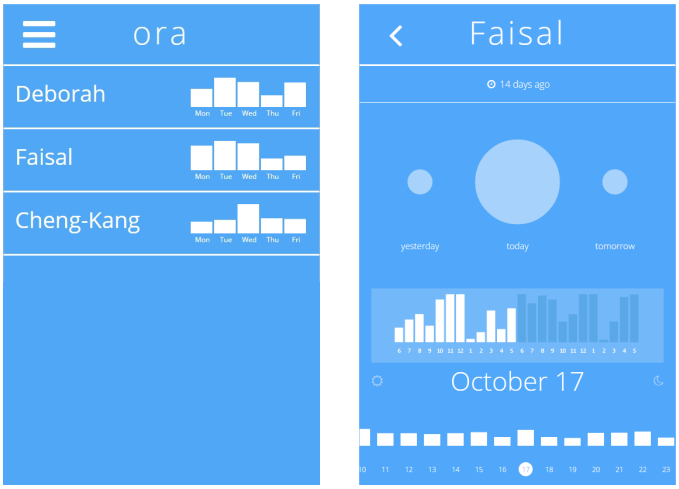
level, PrefMiner [25] monitors on-device notifications from numerous sources to identify which notifications are important to the user or not. Small data differs from these services in its emphasis on providing insights that require longer-term observation, rather than performing transient event-driven actions. This fundamental distinction results in rather different system requirements, particularly in resource management and security as mentioned in the introduction. That said, our small data application architecture could enable a richer set of inputs to both of these systems.

Our aims are similar to existing systems that provide a modular computational infrastructure and mediate the release of processed personal data, such as openPDS and Virtual Individual Servers [26][27]. While these systems do provide personal data acquisition, storage, and release, they do not explicitly address the problem of normalizing and joining disparate data streams under a shared ontology. Our work complements these systems in providing data modeling and interoperability required to join multiple data streams, as opposed to simply providing analysis of individual data streams.

III. Small Data Applications

A small data application is an application that operates on multiple personal data streams, produces some kind of analysis of these streams, and presents the result to the user via an interface. Personal data can include static data, for instance the individual's genome or family lineage. We focus particularly on temporal data, either regular or episodic, that must be continually collected and analyzed. The reason for this focus is twofold: first, these information-rich data sources will be most transformative in creating detailed user models and feedback for diverse applications, and second the temporal data are the more difficult to manage since it is constantly accumulating. Of course, our focus on temporal data does not obviate the value of joining the user's data with other non-temporal data sets - e.g., summarizing nutritional exposure using temporal grocery receipts and relatively-static nutritional databases.

Below, we motivate the requirements of our software architecture using three exemplar small data apps. These applications comprise two data access modes – background and foreground. In the background mode, the application may periodically access a long history of user data to build or update the user's behavioral model. In the foreground, the user experience tends to be based on a more recent window of time, interpreted in the context of the behavioral model.



(a) Ora: List (b) Ora: User Details

Figure 2. Ora: User List and Details View

A. Ora

Ora (Figure 2) is a tool for sharing how you are doing – without sharing the details of what you are doing – with family, friends, or other people who might be part of your support network (counselors, coaches, etc.) Users interact with Ora via a mobile-optimized website, where they authorize the app to connect to their Gmail and Moves accounts using an OAuth2 grant. Ora extracts descriptive numeric features from these data sources and uses them to build a baseline model that represents the user's usual values for each feature. Deviations from this model are calculated on a per-day basis and summarized into a single numeric value, referred to as a *pulse*, that acts an opaque indicator of the degree to which the user is deviating from the model.

Specifically, the pulse is computed from 20 features extracted from the users' data, including their **geodiameter** (the distance between the furthest two points in their location trace for the day), **exercise duration** (the number of minutes the user was walking or running), **time not at home** (the amount of time not spent at their primary location, typically their home), and the **number of emails sent** in a day. Then, for a set of features F, the baseline for each $f \in F$ is computed as a tuple consisting of the sample standard deviation and mean over a two-month sliding window. For a given day, the pulse (P) is then computed as the sum of the numbers of standard deviations from the mean for each feature.

B. Pushcart

Pushcart (Figure 3) uses receipts from services such as FreshDirect or Peapod to determine the nutritional value of the food that a household purchases. This information is provided to a "Wizard of Oz" system in which a clinician, masquerading as a learning algorithm, reviews the purchasing habits of each household and suggests substitutions of more nutritional items during future purchases.

The system's primary source of input is email – after opting in, users register the system to automatically receive a copy of their receipt email, from which the list of items is extracted and then joined against a database of nutritional information for each food item. The user interacts with the system through email as well: the user interface is a weekly "report email"

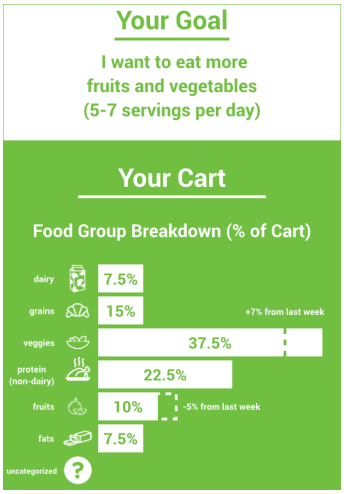


Figure 3. Pushcart: Weekly Email Report

that shows a breakdown of purchases in terms of nutritional value, and includes the nutritionist's suggestions.

C. Partner

Partner is an exploratory app designed around the hypothesis that people who spend time together tend to mimic each others' language patterns, and that the extent of this mimicry is an indicator of good relations; this is a phenomenon known as *linguistic style matching* [28]. The application uses both Gmail and Moves as its data source. After users have registered for the system, it passively collects their email and location data, building a retrospective view of the time they spend physically proximate to each other, the degree of linguistic style matching evidenced by similar values for descriptive metrics used in authorship identification, and the correlation of the two aforementioned values.

Partner relies on a few standard metrics used in authorship attribution, specifically entropy [29], stylometrics such as the percentage of personal pronouns and ratio of functional words to non-functional words (the "information density"), and the index of qualitative variation (IQV, specifically, the Gibbs M1 index) which serves as a measure of the variability of the user's vocabulary. Each of these features is computed over a categorical distribution of the user's tokens, which is produced from the concatenation of a user's emails into week-long intervals to compensate for the sparsity issues that email presents.

IV. Architecture

Our architecture is inspired by the concept of a "mashup", an application that merges multiple disparate data sources into a single interface. We started with the typical web mashup, in which data are acquired, processed, and presented solely by and at the client. We then factored out the acquisition and processing into distinct, reusable modules which can be run in the cloud and potentially consumed by multiple clients. Common concerns, such as caching, access control, and data normalization, are provided as system-wide services. While it would be feasible to implement the acquisition and processing components as tightly-coupled, one-off solutions for a single mashup, the redundancy of doing so for each additional app has lead us toward a centralized and reusable architecture.

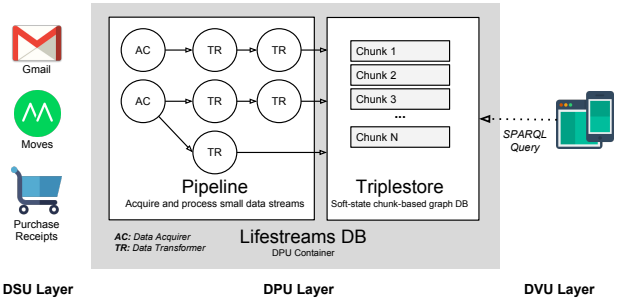


Figure 4. Lifestreams DB Pipeline: consists of a set of DPU modules that acquire and process data from various small data DSUs.

The architecture is composed of three layers, as depicted in Figure 1. There are three main entities: **Data Storage Units** (DSUs), **Data Processing Units** (DPUs), and **Data Visualization Units** (DVUs). These terms mirror the open mHealth standard [30]. DSUs include service provider APIs, e.g., Google's numerous service APIs and Facebook's Graph API. DSUs can be accessed directly from DPUs/DVUs, but are often accessed through a "transforming" DPU that converts the API's often proprietary data format into the schemas we use in small data apps. Data flows from DSUs through arbitrary compositions of DPUs – so long as their input and output types are compatible – and terminates in the DVUs. Lifestreams DB acts as a container for DPUs, and provides caching, data modeling, access control, and a unified query interface. Its outputs can be directly consumed by DVUs, or by other DPUs that provide additional data processing capability.

This modular pipeline approach is necessitated by the fact that our system will never be complete; there will always be new data sources and means of processing and displaying data, which the architecture should readily accommodate. Further, the implementation of its components is a collaborative effort and we wish to encourage developers to reuse and build upon existing components.

V. DPU Containers: Lifestreams DB

Lifestreams DB is an important component in our architecture. Positioned between data sources and small data apps, Lifestreams DB is designed to be the "narrow waist" of the small data ecosystem that provides a unified interface for querying, combining, and fusing diverse small data streams.

Lifestreams DB contains a pipeline of DPUs that Extract, Transform and Load (ETL) an individual's digital traces from different sources using common software APIs and Schemas to enable diverse small data applications. Figure 4 illustrates the architecture of Lifestreams DB. On the left is ***Lifestreams Pipeline***, a data processing pipeline that contains a set of reusable DPUs that extract raw data from different small data sources and transform raw data into structured, readily usable information. For example, raw actigraphy and geolocation sensor samples from a mobile app are transformed into structured data that describe the time, location, speed, and distance of each activity episode. These extracted data are loaded into ***Lifestreams Triplestore***, an RDF datastore built on top of Jena TDB [31], that exposes an integrated view of all the diverse RDF data for apps to query. We made two principal design decisions when designing Lifestreams DB: 1) to model data using RDF, and 2) to utilize a soft-state system design. The rationales behind these design decisions are described in the

 ISBN: 978-1-61208-631-6

following.

a) Using RDF for interoperability: Data interoperability is key to the success of such a system. Raw data extracted from different data silos need to be transformed into a compatible form to allow one to derive knowledge from them. In Lifestreams DB, we utilize RDF to enable data interoperability. Each DPU outputs data in JavaScript Object Notation (JSON), and the DPUs at the final stage generate RDF data in the JSON-LD format, which will be transformed into RDF triples (i.e., *subject-predicate-object*) before stored in the Triplestore. The advantages of using RDF are as follow. First, it eliminates the need to define database schema, unlike, for example, in a Structured Query Language (SQL) datastore. Data generated by different DPUs are inherently interoperable if the DPUs follow the same ontology to model the data. This property is of significant benefit to a small data ecosystem, since it allows DPUs developed by different people to be plug-and-play without the need to modify the system's database schema. Also, any client application developer, given the ontology, can compose queries to filter, join, and aggregate various types of data generated by different DPUs without knowing specific implementation details such as table and column names, etc.

b) A Soft-State System Design: Architecturally, one major difference between an individuals' digital traces and an enterprise's operational data is that an individual's data are mostly persisted and protected in each original data source's databases (e.g., Google, Facebook). In many cases, there is no need, and is actually wasteful and harmful to the users' security and privacy, for Lifestreams DB to replicate all these data in one place. Thus, we propose a soft-state design that, while providing the client applications with virtual access to all the data, only caches a small portion of it in the system. Data which the user owns (e.g., sensor data from the user's phone or wearable) can be considered in the same way, except that it will reside on a personal DSU instead of in an external organization.

The advantages of this design are three-fold: First, a soft-state design requires much less storage to serve the requests, and thus allows the system to scale more effortlessly to serve a larger number of users and integrate with more diverse information beyond its storage capacity. Further, it enables elastic storage provision, where a service provider can provide the service with less storage (at consequently lower cost), and increase the storage provision only when better performance is needed. Second, it makes the system more robust, since there are less points where critical data loss can occur. If the system needs to be brought down, it can be done so without concern over maintaining important state. Third, a soft-state design inherently has better security properties. Since only a small amount of information is cached in the system at any given time, the exposure of any single data breach is limited. In addition, the fact that the data can be repopulated into the database on-the-fly allows us to encrypt sensitive data and only decrypt them when they are demanded.

These advantages do not come without a price. A soft-state system tends to incur much overhead in indexing, reproducing, and reloading data. In Lifestreams DB, we reduce these overheads by utilizing a chunk-based data management strategy that generates and manages data in chunks. Our design is particularly suitable for applications that perform timeseries-based analysis with temporal locality where subsequent accesses tend to access records that are near in time (in our scheme, in the same chunk.) Within these assumptions, we have improved Lifestreams DB's query performance by multiple factors (compared to the base Jena TDB triplestore) and made it perform even better than a hard-state system that stores all the data with only a fraction of storage space.

TABLE I. DATA MODELING TYPE ASSIGNMENTS

Data	Source	Subject Types	Object Types
Location/Mobility	Moves API [32]	Stay/Travel	Place
Email	Gmail API	Send/Receive	EmailMessage
Purchase	Gmail API	Buy	Product
Calendar	gCal API	Join	Event
Web Browse	Android API	Browse	WebPage
App Usage	Android API	Use	MobileApp
Phone Call	Android API	Call/Receive	Person
Message	Android API	Send/Receive	SMSMessage

In the following, we first describe our RDF-based data modeling approaches and demonstrate its advantages using the SPARQL queries for the real-world small data applications we are developing. Then, we describe the chunk-based management strategy and the techniques we used to realize the proposed soft-state design.

A. Data Modeling

When modeling data using RDF, one needs to follow a certain *ontology*. In small data, the concepts we come across most often are the various *actions* performed by users, such as sending emails, making purchases, etc. We chose schema.org [33] as the main ontology rather than the other competing candidates, such as Activity Streams [34], for its semantic action type system. Schema.org defines a hierarchical type system that describes different (sub)categories of actions. At the root is **Action**, a generic type that describes the common properties of an action (e.g., agent, time, etc.). It is then subclassed by more specific types, such as **MoveAction**, which, in turn, are subclassed by more specific types, such as **ArriveAction**, **DepartAction**, etc. This hierarchical structure enables one to write queries to reason across different types of actions within specific categories. For example, an app that encourages better sleep hygiene may analyze users' before-sleep routines by querying certain action categories (e.g., the ExerciseAction and all its subclasses) that occurred before the sleep period.

Table I summarizes eight different kinds of data we have extracted and modeled from four different data sources, based on schema.org's ontology. The purchase records are derived from email receipts on an opt-in basis. The phone-based data are uploaded to ohmage, a mobile sensing DSU. In the following, we demonstrate how our data modeling approaches can satisfy the requirements of the small data applications described previously with simple SPARQL queries.

Ora: Listing 5 shows a snippet of Ora Query that computes the geodiameter and the number of emails sent in a day. For brevity, the snippet omits the part that limits the time range to a single day. The first part of the snippet computes the geodiameter by selecting the maximum distance between any pairs of places at which the user stayed. The second part of the query counts the number of **SendAction**'s of which the targeted object is an email. This example is intended to

```
PREFIX schema: <http://schema.org/>
SELECT * {
  SELECT (MAX(?dist) AS ?geodiameter)
  {  ?stay_x a schema:StayAction;
             schema:location ?loc_x.
     ?stay_y a schema:StayAction;
             schema:location ?loc_y.
     BIND (
        fn:distanceInMeter(?loc_x, loc_y) AS ?dist
    ).}
  SELECT (COUNT(?send) AS ?mail_count)
  {  ?send a schema:SendAction;
           schema:object ?object.
     ?object a schema:EmailMessage.}
}
```

Figure 5. A short snippet from Ora query that computes the geodiameter and the number of emails sent.

```
PREFIX text: <http://jena.apache.org/text#>
PREFIX usda: <http://data-gov.tw.rpi.edu/vocab/p/1458/>
SELECT *
{  ?action a schema:BuyAction.
   ?action schema:object ?product.
   ?product schema:name ?product_name.
   SERVICE <http://localhost/usda/endpoint> {
    ?food_item text:query
        (usda:shrt_desc ?product_name 1).
    ?food_item usda:carbohydrt ?carbon;
               usda:protein ?protein.
   }
}
```

Figure 6. Pushcart Query joins an individual's food purchase records with the corresponding nutritional information contained in the USDA nutrient database.

demonstrate how much an application developer can achieve with Lifestreams DB using a succinct and easy to understand query. Also, this example demonstrates how heterogeneous data streams (i.e., Location/Mobility and Email) are modeled and queried in an interoperable and standardized way.

Pushcart: Listing 6 shows a snippet of the Pushcart query. It demonstrates Lifestreams DB's interoperability with an external food nutrition database. A RDF dump of the United States Department of Agriculture (USDA) nutrient database is pre-loaded into a separate triplestore [35]. The query joins the individuals' grocery purchase records with the entries contained in the USDA database using a free-text matching based on the product names, and select the amount of carbohydrates and protein contained in each of the purchased items.

Partner: Partner is an example of an app which, in addition to Lifestreams DB, requires a more domain-specific DPU. It relies on Lifestreams DB to compute the amount of time two participants spent together based on the distance between where two users stay (see Listing 7) and uses the Email Analysis Framework (EAF), a DPU for email language analysis [36], to evaluate language style matching. It is also an example where an application can query from not only one but across multiple users' data with RDF *named graphs* that refer to each user.

```
PREFIX fn: <http://lifestreams.example.org/customFn#>
PREFIX users: <http://lifestreams.example.org/users#>
SELECT (SUM(?overlap) AS ?co_present_time)
{  GRAPH <users:Bob> {
      ?stay_x a schema:StayAction;
      schema:location ?loc_x.}
   GRAPH <users:Alice> {
      ?stay_y a schema:StayAction;
      schema:location ?loc_y.}
   FILTER(fn:distanceInMeter(?loc_x, ?loc_y) < 50)
   BIND (
    fn:overlappingTime(?stay_x, ?stay_y) AS ?overlap
   )
}
```

Figure 7. Partner Query computes the time two users spent together based on their location data. Each user's data are referred to by their *named graph*.

B. Chunk-based Data Management

As mentioned, Lifestreams DB's soft-state design is made possible by a chunk-based strategy. The basic idea behind this strategy is as follows: The DPUs in Lifestreams Pipeline generate data in chunks and load them into Lifestreams Triplestore, which maintains an index to all the chunks (including the ones that are not cached in the system). When a client application submits a query, it will additionally submit a meta-query that selects the chunks it desires. If a chunk selected by the meta-query is not currently available in the system, Lifestreams Pipeline will re-run the corresponding DPUs and reproduce the chunk on the fly from the source. The chunks that contain sensitive data (determined from the data source and the user's preferences) will be encrypted and decrypted on the fly when requested by a query. The chunks are encrypted with 256-bit Advanced Encryption Standard (AES).

Our strategy allows a system to maintain only a small amount of information (i.e., the chunk index) while providing access to much larger amount of data that is beyond the system's storage capacity. In the following, we describe three major designs that realize this strategy and discuss several query optimization techniques enabled with chunking that can be utilized to provide a better user experience.

1) Chunk Index Design: The chunk index needs to be carefully designed to avoid unnecessary chunk reproduction. For each chunk of data, we extract the following features as its index:

- Distinct object types in the chunk.
- Start time and end time of the aggregate timespan.
- Geo-coordinates of a convex hull that covers all the spatial features in the chunk.

The rationales behind these choices are as follow. First, most of our applications are interested in certain types of actions or objects (e.g., CommunicationActions or ExerciseActions) so object types are a natural choice for indexing. Also, most of small data are time-tagged, and the applications we focus on tend to involve analysis of time series and aggregation based on time or location. Therefore, it is important for us to make chunk index satisfy these requirements.

2) Lifestreams Pipeline: a reproducible pipeline: We adopt a functional approach to allow Lifestreams Pipeline to reproduce arbitrary chunks of data from the original sources. The Lifestreams Pipeline consists of two types of DPUs: **Acquirers** acquire raw data from the sources while **Transformers**

transform data from one form to another. These DPUs are treated as passive functions invoked by the system. Consider a simple pipeline where one Acquirer and one Transformer linked in sequence. In each iteration, the system invokes the Acquirer with a *state variable* that indicates the chunk we want the Acquirer to fetch. After fetching the corresponding chunk, the Acquirer will return the chunk along with a new state variable that indicates the subsequent chunk to be acquired in the next iteration. The system then invokes the Transformer to transform the chunk, and stores the output chunk along with the state variable. When the chunk is removed, the state variable will be preserved in the system. Therefore, when we need to reproduce the chunk, we just need to re-run the pipeline with the preserved state variable.

An assumption we make here is that the raw data are permanently persisted in the original data sources (i.e., DSUs), and can be re-acquired by the Acquirer anytime. If this is not the case, a *shim* can be implemented to transfer the data to a DSU with such properties (such as Amazon S3). Unlike some chunk-based systems where the chunk sizes are pre-determined, Lifestreams DB allows each Acquirer to decide the chunk sizes according to the characteristics of the APIs it acquires data from. A typical chunk size is daily as it is supported by most data sources. However, as the state variable is updated by the Acquirers themselves, Acquirers can have state variables with different formats or granularity (e.g., hours, weeks.). This feature is important for small data where one usually needs to work with a large variety of external data sources whose APIs it has no control over.

3) Two-Level GDS Chunk Replacement Policy: Similar to many cache systems, Lifestreams DB requires a replacement policy to select chunks for replacement when the available space is low. Our replacement policy minimizes the overall expected query latency by selecting the chunks that are of larger size and less likely to be used again, and can be reproduced in shorter time. There are two ways to make space in Lifestreams DB: (1) compress the chunk, or (2) evict the chunk entirely. Compression on average results in 7.2x size reduction and can be restored more efficiently than reproducing a chunk from the source. Considering this difference, as well as, the varying chunk sizes and cost in reproducing different kinds of chunks (see Table II), we develop a Two-level Greedy-Dual-Size (Two-Level GDS) replacement policy that is both cost- and size-aware and appropriately choose between two space reduction methods. The basic Greedy-Dual (GD) algorithm assigns each chunk a cost value H. Every time when a replacement needs to be made, the chunk with the lowest H value H_{min} will be replaced first, and all the other chunks reduce their H values by H_{min}. Only when a chunk is accessed again will its H value be restored to its initial value. Greedy-Dual-Size (GDS) incorporates the different chunk sizes by assigning H as $cost/size$ of the chunk [37]. On top of that, our Two-Level GDS algorithm additionally considers the different characteristics of compression and eviction. When a chunk is first inserted into the cache, its $cost$ is set to the estimated decompression latency, and the $size$ is the estimated space reduction after compression. When this chunk is selected for replacement, it will be compressed and re-inserted into the cache with its $cost$ increased to the estimated latency to reproduce it from the source, and the $size$ decreased to its size after compression. Only when this chunk is selected again will it be completely evicted. Similarly, after a chunk is reproduced, it will be first stored in its compressed form. When it is accessed again, it will have a certain probability to be promoted to its decompressed form. The default probability for a compressed chunk to be restored is 0.2. In this way, our algorithm uses compression as the default to make space for its efficiency, but still removes the compressed chunks to reduce cache clutter if they have not been used for long.

TABLE II. GMAIL AND MOVES DATA-SIZE AND REPRODUCTION-TIME CHARACTERISTICS

Avg. Values of 180 Chunks	Gmail	Moves
Chunk Size (KB)	20.32	392.44
Compressed Chunk Size (KB)	3.08	54.12
Required HTTP Requests	14.24	1
Reproduction Time (msec)	1423.63	182.17

4) Chunk-Assisted RDF Query Evaluation: The flexibility of RDF is not without its drawbacks: compared to many SQL datastores, a RDF datastore tends to be slower in query evaluation due mainly to the difficulty of constructing an effective data index [38]. Our chunk-based strategy has several desirable side benefits that mitigate this problem. First, chunk indexes can be utilized as a multi-column index that allows the query engine to take a short path by skipping those data that do not belong to the requested chunks. Second, chunking enables a more effective result cache, which caches the query results and returns the result when the same query is given. Unlike a record-based system, where any modification can potentially invalidate a cached result [38], a chunk-based system only needs to track the modifications of the chunks that generate a cached result to ensure the result's validity. This technique is particularly effective in our system, as most chunks won't change after they have been generated.

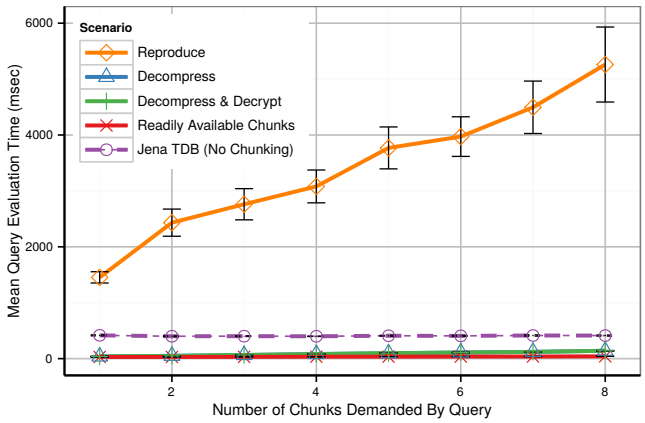
VI. PERFORMANCE EVALUATION

In this section, we evaluate the feasibility and performance of our system using Gmail and Moves data. Using Jena TDB as a baseline, we first evaluate the system performance in different scenarios and with different kinds of data. Then, we evaluate the overall system performance with a real-world query with a workload simulation based on an assumed application usage. The experiment was conducted on an Amazon Web Services (AWS) instance with 8 Intel Xeon E5-2680 processors and 15GB of memory.

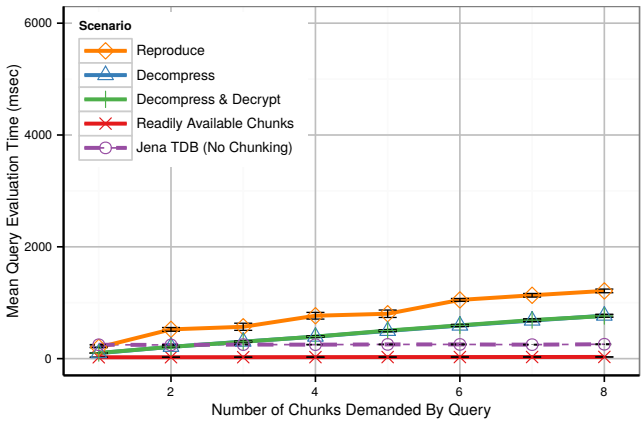
A. Dataset

A dataset of 180 days worth of Gmail and Moves data is used to evaluate the system performance. The data are from three authors of this paper who are regular users of these services. There are in total 360 chunks in the dataset, each of which contains a single day's Gmail or Moves data. Table II summarizes the different characteristic of Gmail and Moves data. For example, while smaller in size, a Gmail chunk requires many more HTTP requests to be issued thus has longer (re)production time. A Moves chunk, on the other hand, can be (re)produced in a much shorter time, but usually is much larger in size due to the high-resolution location traces. These differences will result in different performance characteristics as shown in the following. These differences must be taken into account to achieve efficient resource utilization.

 ISBN: 978-1-61208-631-6



(a) Gmail Data



(b) Moves Data

Figure 8. Query Performance of Different Scenarios: our approach outperforms the Jena TDB by up to 14x when the chunks are readily available. Decompressing is much faster than reproducing a chunk, while decryption adds only negligible overhead. Gmail data requires more time to reproduce since more HTTP requests need to be made. The varying performance in different scenarios evidence the need of an cost- and size-aware chunk replacement policy.

B. *Query Performance*

We compare the query performance of our system with our baseline, Jena TDB, based on the following scenarios:

1) The demanded chunks are readily available.
2) The chunks need to be decompressed.
3) The chunks need to be decompressed and decrypted.
4) The chunks need to be reproduced from the data source.

The results suggest up to 14x performance improvement over Jena TDB for a both a simple query and a complex real-world query. The experiment was conducted with all 360 chunks pre-loaded into the triple store. Each data point presented below is an average of 30 runs of the experiment. The error bars in the figures are the 95% confidence interval.

1) Simple Query Performance: We first evaluate the performance with a simple query that counts the number of distinct Action subjects. Figure 8a and Figure 8b show the results for Gmail and Moves data respectively, where the x-axis is the number of chunks demanded in the query, and the y-axis is the mean query evaluation time. When the demanded chunks are cached in the system, our system outperforms Jena TDB by up to 14x and 10x for Gmail and Moves respectively. This performance gain is mainly attributed to the chunk-skipping optimization mentioned in the Chunk-Assisted Evaluation section. For Gmail data, decompressing shows up to 36x better performance than reproducing, and decryption adds only negligible overhead (less than 1.3%). This difference is not that significant for Moves, since Moves data can be reproduced in a relatively shorter time, but incurs larger overhead to be inserted into the triplestore in either scenario.

2) Real-World Query Performance: Next, we use a real-world query to demonstrate the system performance in a more realistic setting. A query from one of our small data applications, Ora, is used. It consists of 211 lines of SPARQL script, extracting 20 features from Gmail and Moves data (See Application section). Since this more complex query requires a larger number of scans to be made over the search space, as shown in Figure 9, the performance gain of our chunk-skipping technique becomes more evident (up to 14x improvement over Jena TDB). In addition, due to the longer overall query time, the overhead in decompression and decryption becomes less significant. Reproducing is still the slowest among the four scenarios, but it still outperforms Jena TDB by up to 1.8x.

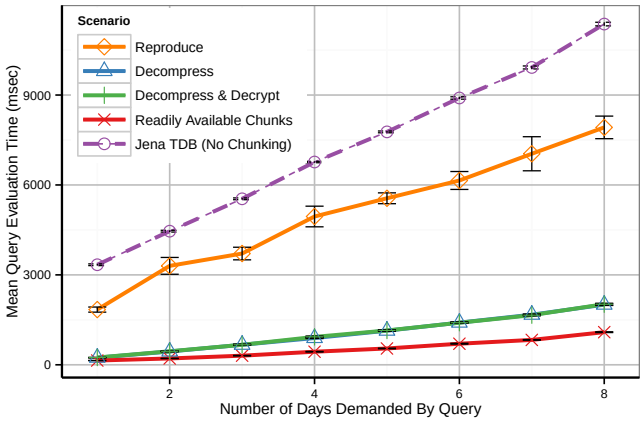


Figure 9. Real-World Query Performance: the performance gain of our chunk-skipping technique becomes more evident (up to 14x) for a complex real-world query where more scans need to be made over the search space.

C. *Performance with Simulated Workload*

The varying performance for different types of data and scenarios stresses the need for a chunk replacement policy that is able to incorporate these discrepancies. We evaluate the effectiveness of the proposed Two-Level GDS algorithm using a simulated workload of Ora. Based on the UI of Ora, we assume a binomial process usage pattern where each page shows one-week worth of data and can be browsed in a reverse chronological order. We assume the user will use the app daily, and after viewing a page, the user has a probability p to browse the next page or a probability of $1-p$ to leave the app. We set $p = 0.7$ and compare our approach with well-known Least-Recently-Used (LRU) policy, as well as the Jena TDB that

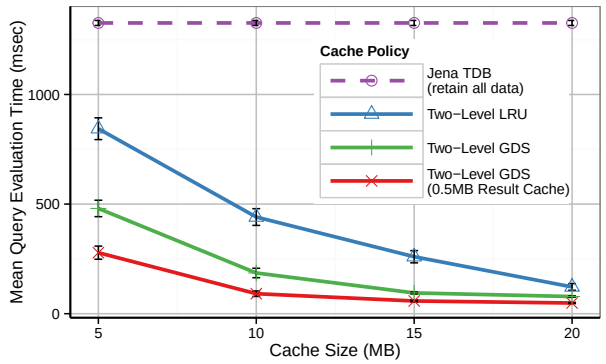


Figure 10. Query Performance with Simulated Workload: our Two-Level GDS approach shows superior performance over LRU, and outperforms Jena TDB that retains all 50.44MB of data, by up to 4.7x using only about 1/10th the storage.

retains all the data. The results suggest that overall, our system outperforms LRU and Jena TDB by up to 4.7x using only a fraction of storage.

We generate 120 days worth of data for the workload based on an assumed usage pattern of Ora. We only consider the performance of the last 60 days when the cache space has become saturated. To allow a fair comparison, we modify the traditional LRU in a way that the chunk chosen for replacement will be first compressed and re-inserted into the LRU list. Only if it is chosen again will it be entirely evicted. We refer to this variant of LRU as *Two-Level LRU*. In addition, for the baseline, Jena TDB, we assume it retains all the 120-day worth of data in the system, which is 50.44MB in size.

Figure 10 shows the performance of different approaches with cache sizes varying from 5MB to 20MB. Our Two-Level GDS shows superior performance over Two-Level LRU especially with a smaller cache size. This advantage comes from the fact the our approach takes the cost of different space reduction methods, and the size of each individual chunk into account. For example, our approach tends to evict a Moves chunk for its shorter reproduction time and larger size. On top of that, if we use 0.5MB of the cache space to cache the query results, we see another 2x of performance improvement. Overall, our approach achieves up to 4.7x performance improvement over Jena TDB, using only about 1/10th the storage. Such a performance improvement is important for small data services to be provided effectively and affordably.

VII. Conclusion and Future Work

In this work, we introduce the notion of small data apps, and the increasing opportunity of these apps to produce deeper and more comprehensive insights across the union of a user's available data, and across a wide range of ubiquitous computing applications. By virtue of the fact that these apps leverage the user as the common denominator and benefactor, there is both the potential for deeper, more personal insights, as well as the need for a robust infrastructure for accessing such intimate data. We present an architecture to support these small data apps that decouples the data sources from the processing and visualization layers, and accounts for the unique challenges presented by contending with sensitive streaming spatio-temporal data from multiple providers. We describe our implementation of a critical component of this architecture, Lifestreams DB, and several candidate applications built on top of it.

Lifestreams DB includes several improvements over existing RDF datastores in terms of storage requirements and query latency, which are likely attributable to the constraints of our domain (i.e., streaming spatio-temporal data which can be reproduced at a cost in latency from an external source.) The application of chunking to the datastore, and a cache eviction policy that leverages both the cost of reproduction/compression and the size of the data, is demonstrated to improve query latency for both a few candidate queries and in a simulated experiment modeling a user's long-term interaction with Ora, an SDA application.

While this work proposes a soft-state architecture to ameliorate the impact of a breach, there is still much work to be done in **secure data storage and distribution** so that breaches are diminished or, preferably, eliminated in the first place. On a related note, there are many improvements that can be made to ensure that the processed data does not compromise the raw data source, and to selectively control who can consume processed data in the case that it is sensitive.

Small data apps address the converse of the big data problem: rather than drawing insights about populations across broad swaths of data for purposes of similar scale (e.g., corporate, governmental, etc.), they draw insights about the individual across their own small data for personal growth and understanding. This work aspires to **foster the growth of the small data ecosystem and the role of small data in fueling ubiquitous computing applications**.

References

[1] S. Perez, "Mobile Application Stores State of Play," 2010. [Online]. Available: http://readwrite.com/2010/02/22/the_truth_about_mobile_application_stores (accessed on 2018.03.19).

[2] H. Halpin, "Social Web Working Group Charter," 2014. [Online]. Available: http://www.w3.org/2013/socialweb/social-wg-charter (accessed on 2018.03.19).

[3] G. Linden, B. Smith, and J. York, "Amazon.Com Recommendations: Item-to-Item Collaborative Filtering," IEEE Internet Computing, vol. 7, no. 1, Jan. 2003, pp. 76–80.

[4] D. Cosley, D. Frankowski, L. Terveen, and J. Riedl, "SuggestBot: Using Intelligent Task Routing to Help People Find Work in Wikipedia," in Proceedings of the 12th International Conference on Intelligent User Interfaces, ser. IUI '07. New York, NY, USA: ACM, 2007, pp. 32–41.

[5] "Foursquare." [Online]. Available: https://foursquare.com/ (accessed on 2018.03.19).

[6] Z. Gantner, S. Rendle, C. Freudenthaler, and L. Schmidt-Thieme, "Mymedialite: A free recommender system library," in Proceedings of the fifth ACM conference on Recommender systems. ACM, 2011, pp. 305–308.

[7] M. D. Ekstrand, M. Ludwig, J. Kolb, and J. T. Riedl, "Lenskit: a modular recommender framework," in Proceedings of the fifth ACM conference on Recommender systems. ACM, 2011, pp. 349–350.

[8] M. Swan, "The Quantified Self: Fundamental Disruption in Big Data Science and Biological Discovery," Big Data, vol. 1, no. 2, Jun. 2013, pp. 85–99.

[9] FitnessKeeper, Inc., "Health Graph API," 2014. [Online]. Available: http://developer.runkeeper.com/healthgraph/ (accessed on 2018.03.19).

[10] S. Consolvo, D. W. McDonald, T. Toscos, M. Y. Chen, J. Froehlich, B. Harrison, P. Klasnja, A. LaMarca, L. LeGrand, R. Libby, I. Smith, and J. A. Landay, "Activity Sensing in the Wild: A Field Trial of Ubifit Garden," in Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, ser. CHI '08. New York, NY, USA: ACM, 2008, pp. 1797–1806.

[11] M. Lin, N. D. Lane, M. Mohammod, X. Yang, H. Lu, G. Cardone, S. Ali, A. Doryab, E. Berke, A. T. Campbell, and T. Choudhury,

"BeWell+: Multi-dimensional Wellbeing Monitoring with Community-guided User Feedback and Energy Optimization," in Proceedings of the Conference on Wireless Health, ser. WH '12. New York, NY, USA: ACM, 2012, pp. 10:1–10:8.

[12] J. Costa, A. T. Adams, M. F. Jung, F. Guimbetiere, and T. Choudhury, "Emotioncheck: leveraging bodily signals and false feedback to regulate our emotions," in Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing. ACM, 2016, pp. 758–769.

[13] W. Jones, "Personal information management," Annual review of information science and technology, vol. 41, no. 1, 2007, pp. 453–504.

[14] L. Sauermann, G. A. Grimnes, M. Kiesel, C. Fluit, H. Maus, D. Heim, D. Nadeem, B. Horak, and A. Dengel, "Semantic Desktop 2.0: The Gnowsis Experience," in The Semantic Web - ISWC 2006, ser. Lecture Notes in Computer Science, I. Cruz, S. Decker, D. Allemang, C. Preist, D. Schwabe, P. Mika, M. Uschold, and L. M. Aroyo, Eds. Springer Berlin Heidelberg, Jan. 2006, no. 4273, pp. 887–900.

[15] Y. Cai, X. L. Dong, A. Halevy, J. M. Liu, and J. Madhavan, "Personal Information Management with SEMEX," in Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data, ser. SIGMOD '05. New York, NY, USA: ACM, 2005, pp. 921–923.

[16] B. Salmon, S. W. Schlosser, L. F. Cranor, and G. R. Ganger, "Perspective: Semantic Data Management for the Home," in Proccedings of the 7th Conference on File and Storage Technologies, ser. FAST '09. Berkeley, CA, USA: USENIX Association, 2009, pp. 167–182.

[17] T. Gupta, R. P. Singh, A. Phanishayee, J. Jung, and R. Mahajan, "Bolt: Data Management for Connected Homes," in Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation, ser. NSDI'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 243–256.

[18] Rustici Software, "xAPI." [Online]. Available: https://xapi.com/ (accessed on 2018.03.19).

[19] Apple Inc., "HealthKit." [Online]. Available: https://developer.apple.com/healthkit/ (accessed on 2018.03.19).

[20] Microsoft, "HealthVault." [Online]. Available: https://www.healthvault.com/ (accessed on 2018.03.19).

[21] Epic System Corp., "MyChart." [Online]. Available: https://mychart.deancare.com/mychart/ (accessed on 2018.03.19).

[22] "Google Fit." [Online]. Available: https://fit.google.com/ (accessed on 2018.03.19).

[23] M. Van Kleek, B. Moore, D. R. Karger, P. André et al., "Atomate it! end-user context-sensitive automation using heterogeneous information sources on the web," in Proceedings of the 19th international conference on World wide web. ACM, 2010, pp. 951–960.

[24] IFTTT, "IFTTT: Put the internet to work for you." 2014. [Online]. Available: https://ifttt.com/ (accessed on 2018.03.19).

[25] A. Mehrotra, R. Hendley, and M. Musolesi, "Prefminer: mining user's preferences for intelligent mobile notification management," in Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing. ACM, 2016, pp. 1223–1234.

[26] Y.-A. de Montjoye, E. Shmueli, S. S. Wang, and A. S. Pentland, "openpds: Protecting the privacy of metadata through safeanswers," PloS one, vol. 9, no. 7, 2014.

[27] A. Shakimov, H. Lim, R. Cáceres, L. P. Cox, K. Li, D. Liu, and A. Varshavsky, "Vis-a-vis: Privacy-preserving online social networking via virtual individual servers," in Communication Systems and Networks (COMSNETS), 2011 Third International Conference on. IEEE, 2011, pp. 1–10.

[28] K. G. Niederhoffer and J. W. Pennebaker, "Linguistic Style Matching in Social Interaction," Journal of Language and Social Psychology, vol. 21, no. 4, Dec. 2002, pp. 337–360.

[29] M. Grabchak, Z. Zhang, and D. T. Zhang, "Authorship Attribution Using Entropy," Journal of Quantitative Linguistics, vol. 20, no. 4, Nov. 2013, pp. 301–313.

[30] Open mhealth developer wiki. (accessed on 2018.03.19). [Online]. Available: https://github.com/openmhealth/developer/wiki [retrieved: April, 2014, accessed on 2018-03-18]

[31] The Apache Software Foundation, "Apache Jena," 2014. [Online]. Available: http://jena.apache.org/ (accessed on 2018.03.19).

[32] "Moves API." [Online]. Available: https://dev.moves-app.com/ (accessed on 2018.03.19).

[33] Schema.org Community Group, "Schema.org core schema," 2018. [Online]. Available: http://schema.org/docs/schema_org_rdfa.html (accessed on 2018.03.19).

[34] J. M. Snell, "Activity Streams 2.0," 2015. [Online]. Available: http://www.w3.org/TR/activitystreams-core/ (accessed on 2018.03.19).

[35] USDA, "National Nutrient Database for Standard Reference." [Online]. Available: http://data-gov.tw.rpi.edu/wiki/Dataset_1458 (accessed on 2018.03.19).

[36] F. Alquaddoomi, C. Ketcham, and D. Estrin, "The Email Analysis Framework: Aiding the Analysis of Personal Natural Language Texts," in Hypertext 2014 Extended Proceedings, ser. CEUR Workshop Proceedings, F. Cena, A. S. d. Silva, and C. Trattner, Eds., vol. 1210. CEUR-WS.org, 2014.

[37] P. Cao and S. Irani, "Cost-Aware WWW Proxy Caching Algorithms," in Proceedings of the USENIX Symposium on Internet Technologies and SystemsMonterey, California, December 1997, 1997, pp. 193–206.

[38] M. Martin, J. Unbehauen, and S. Auer, "Improving the Performance of Semantic Web Applications with SPARQL Query Caching," in The Semantic Web: Research and Applications, ser. Lecture Notes in Computer Science, L. Aroyo, G. Antoniou, E. HyvÃűnen, A. t. Teije, H. Stuckenschmidt, L. Cabral, and T. Tudorache, Eds. Springer Berlin Heidelberg, Jan. 2010, no. 6089, pp. 304–318.