UPnQ: an Architecture for Personal Information Exploration

Sabina Surdu[1], Yann Gripay[1], François Lesueur[1], Jean-Marc Petit[1], and Romuald Thion[2]

[1] INSA-Lyon, LIRIS, UMR5205
F-69621, France
firstname.lastname@liris.cnrs.fr
[2] Université Lyon 1, LIRIS, UMR5205
F-69622, France
firstname.lastname@liris.cnrs.fr

Abstract. Today our lives are being mapped to the binary realm provided by computing devices and their interconnections. The constant increase in both amount and diversity of personal information organized in digital files already turned into an information overload. User files contain an ever augmenting quantity of potential information that can be extracted at a non-negligible processing cost. In this paper we pursue the difficult objective of providing easy and efficient personal information management, in a file-oriented context. To this end, we propose the Universal Plug'n'Query (UPnQ) principled approach for Personal Information Management. UPnQ is based on a virtual database that offers query facilities over potential information from files while tuning resource usage. Our goal is to declaratively query the contents of dynamically discovered files at a fine-grained level. We present an architecture that supports our approach and we conduct a simulation study that explores different caching strategies.[3]

Keywords: files information overload, personal information management, potential information, declarative file querying, wrappers

1 Introduction

Computers have triggered a significant paradigm shift in our lives. Our daily existence is mapped to the binary realm provided by interconnected computing devices. Everyday actions, events, activities are now translated into personal data, most of them being organized in personal files: personal maps of morning joggings in the park, digital playlists of songs from our parents' vinyl record collections, personal health & social care information in Electronic Health Record systems, *etc.* Interactions between citizens and public administration agencies are being progressively dematerialized too [6].

Our research is motivated by the drawbacks of current personal file data management technology. There is a large amount of heterogeneous files storing personal data

[3] This work is partially funded by the KISS Project (ANR-11-INSE-0005) of the French National Research Agency (ANR).

such as videos, music, semi-structured documents, images, etc. There is also a growing number of technologies that can extract interesting knowledge from these files: image and video processing, data and text mining, machine learning, speech recognition, image pattern recognition, musical analysis, *etc.* Managing user files in this context becomes cumbersome for both developers and end-users. Developers encounter time-consuming difficulties to write applications handling heterogeneous files, while end-users find it difficult to search through all their data to retrieve useful information. Dedicated Personal Information Management (PIM) systems can ease information management on specific domains but usually lack file content querying capabilities. In our previous benchmark [12], we show how data-oriented pervasive application development gets significantly simplified when using declarative queries. Yet, to the best of our knowledge, there are no declarative query languages that can provide application developers with the ability of writing homogeneous queries over heterogeneous user files, that could be the core building block of powerful file-oriented personal data management applications.

We want to be able to pose fine-grained declarative queries at different levels of granularity: structure, metadata and content. We identify the following main challenges: (1) structure heterogeneous files into a homogeneous model; (2) provide a homogeneous interface for different data extraction technologies; (3) use an efficient query execution strategy allowing to query large amounts of files as soon as possible; (4) design a high level declarative query language, abstracting file access issues and enabling optimization techniques.

This paper introduces the UPnQ user files data management approach: a *virtual* database that offers query facilities over *potential data* from files that can be obtained on demand by dedicated wrappers. To this end, we propose a homogeneous representation of heterogeneous files, regardless of their file format, and fine-grained query facilities: a *UPnQ file* is a *physical file* viewed through a *wrapper* that understands its format and semantics. Like SQL over relational DBMSs for most applications, UPnQ is designed to ease application development rather than to serve as a direct end-user interface. This work has been carried out in the framework of the ongoing KISS project[4], devoted to managing personal information on secure, portable embedded devices called Personal Data Servers (PDSs).

The *file model* and the *query language* are introduced in Section 2. The architecture of the UPnQ system is detailed in Section 3. Different query processing strategies are evaluated in Section 4. Section 5 positions our approach with respect to related work.

2 Model and Language

To easily build applications over physical files, we first propose a homogeneous data-centric representation of such files in non-first normal form. Subsequently, we define an SQL-like file-oriented query language, the *UPnQ file query language*. The full expressiveness of SQL can then be applied when querying data from files: join queries can examine related data from different files, aggregate queries can compute statistics over a set of files, *etc.*

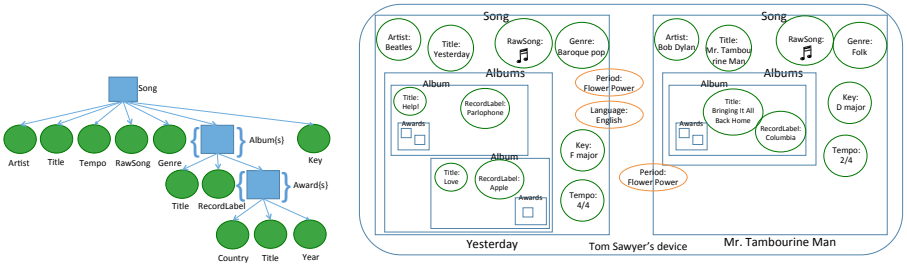
[4] https://project.inria.fr/kiss/en

2.1 Data Model

Data from UPnQ files are hierarchically organized in *capsules* and *seeds*. The outermost capsule of a file holds all the data from the file. A capsule can contain seeds and / or lists of inner capsules. Seeds provide atomic data. The capsule-and-seeds terminology reflects the (nested) non-first normal form relational representation of files: it enables a coherent description of heterogeneous data in different files.

The schema of a capsule can be described by a tree data structure, whose root is the capsule's name. Figure 1(a) shows the schema of a $Song$ capsule. The root node and intermediary nodes (squares on the figure) are capsule names, and leaf nodes (circles) are seed names. Intermediary nodes shown between curly braces represent lists of capsules at the data level, i.e., the instance level of a UPnQ file. Within a capsule, seeds receive atomic values, e.g., $Artist$ might have a value of "Beatles", *etc.*

The type of a file is determined by the name of the outermost capsule. A file can also be individually associated with a set of tags, that are usually user-defined and provide further information about the file as seen through the user's eyes. Figure 1(b) shows two UPnQ files (*Yesterday* and *Mr. Tambourine Man*) on user Tom Sawyer's device. Both files have a $Song$ file type, due to their outermost capsule. They also contain atomic data at the $Song$ capsule level with seeds $Artist$, $Title$, $Tempo$, *etc.* The $Song$ capsule also contains a list of $Album$ capsules. Each $Album$ capsule describes the $Title$ and $RecordLabel$ of the album, and inner capsules for $Awards$ earned by the album.



(a) The schema of a song capsule (b) Two song files on Tom Sawyer's device

Fig. 1. Schema and Instances of Song capsules

2.2 Query Language

We want to express fine-grained queries on data from files sharing a common structure of interest. The query engine is responsible for selecting files providing wanted capsules, and for extracting capsule data from files through corresponding wrapper calls. We extend the SoCQ system [10] to this purpose; the resulting engine allows us to perform fine-grained, lazy data extraction from files, in a relational setting, using SoCQ's *binding patterns* and *virtual attributes* constructs to build XD-Relations (eXtended Dynamic Relations). For UPnQ, SoCQ's concept of SERVICE is replaced with CAPSULE

to interact with wrappers through binding patterns in order to provide values for virtual attributes. Our goal is to achieve the UPnQ vision with a UPnQ file engine thanks to the core mechanism of the SoCQ system.

A relational view on file sets We define a *capsules translation* operator that builds a XD-Relation representing a set of files providing some common features. The purpose of this operator is to translate capsules into a relational, SoCQ-queryable view. We also define a *tags translation* operator, that builds a relation with values for a set of tags associated to a set of files.

Listing 2(a) describes the UPnQ query that creates an XD-Relation based on music files providing $Artist$, $Title$, Key and $Album$ capsules. Assume the file repository contains three songs. The resulting XD-Relation $FPSongs$ is shown in Figure 2(c). There are three capsules that contain capsules $Artist$, $Title$, Key and $Album$, all of them being $Song$ capsules in this case. We have therefore one tuple for each $Song$ capsule. Data are however not materialized yet, as attributes are virtual in this XD-Relation.

Querying files in UPnQ The next step is to query files' capsules using the binding patterns associated to XD-Relations to retrieve data for virtual attributes. Using previous XD-Relation $FPSongs$, Listing 2(b) describes a UPnQ query that materializes data concerning the Artist, song Title, and the Album Title. Through binding patterns, a query can interact with wrapper invocations and control their behavior. The resulting XD-Relation is depicted in Figure 2(d). Some data, here the albums' record label, do not need to be retrieved from the files for this query.

```
CREATE RELATION FPSongs (
  Song CAPSULE,
  Artist STRING VIRTUAL,
  Title STRING VIRTUAL,
  Album CAPSULE VIRTUAL,
  Album.Title STRING VIRTUAL,
  Album.RecordLabel STRING VIRTUAL )
USING BINDING PATTERNS (
  Artist[Song] ( ) : ( Artist ),
  Title[Song] ( ) : ( Title ),
  Album[Song] ( ) : ( Album,
    Album.Title, Album.RecordLabel ) )
AS TRANSLATE CAPSULES
  PROVIDING Artist, Title, Key, Album
```

(a) a UPnQ Capsules Translation Query

```
SELECT Artist, Title, Album.Title
FROM FPSongs
WHERE Title = "Mr. Tambourine Man"
  or Title = "Yesterday"
USING BINDING PATTERNS
  Artist, Title, Album;
```

(b) a UPnQ Data Extraction Query

Song	Artist	Title	Album	Album.Title	Album.RecordLabel
music: Yesterday.mp3	*	*	*	*	*
music: MrTambourineMan.ogg	*	*	*	*	*
music: DearPrudence.mp3	*	*	*	*	*

(c) XD-Relation result of the UPnQ Capsules Translation Query ($\star$ is absence of value)

Artist	Title	Album.Title
Bob Dylan	Mr. Tambourine Man	Bringing It All Back Home
Beatles	Yesterday	Help!
Beatles	Yesterday	Love

(d) XD-Relation result of the UPnQ Data Extraction Query for two songs

Fig. 2. Examples of UPnQ queries – Capsules Translation & Data Extraction

3 Enabling Technologies and Architecture

UPnQ Wrappers. The UPnQ system relies on *parameterizable wrappers* that can be invoked to extract pieces of data from various physical files and expose them, in a homogeneous manner. Dynamic *UPnQ wrappers* present two innovative features: 1) they allow fine-grained data management, and 2) they can be invoked on files on demand, i.e., when a query requires data from a physical file.

There's a multitude of APIs out there that allow manipulating various file formats. The iCal4j API allows the manipulation of iCalendar files, the Apache POI API provides support for Microsoft Office files, *etc.* The development of such APIs supports the fact that our UPnQ wrapper vision is feasible. For these wrappers, we envision an ecosystem of downloadable plugins, like today's application stores.

UPnQ Queries. UPnQ files are mapped to relational tuples. Some of the reasons for choosing to project a file environment on the relational canvas can be drawn from [9]: the simplicity, the robustness and the power of the model, which are well-known in the database community for a long time.

The goal of the UPnQ system is to express declarative queries on top of UPnQ files, so queries need to be given control over imperative wrappers. We also need to provide a virtual database over files, where data can be extracted using both lazy and eager approaches. We turn our attention to Pervasive Environment Management Systems (PEMSs) like SoCQ [10] or Active XML [3]. Both systems can homogeneously query distributed heterogeneous entities and allow lazy evaluation of remote service methods. SoCQ is here preferred to Active XML for its SQL-like language.

Architecture. Figure 3 shows the main components of the UPnQ system.The *UPnQ file system* manages the UPnQ wrappers and files, i.e., the available wrappers and files in the system. The *UPnQ relational file engine* contains a query engine that handles relations, smart folders (subset of files defined by dedicated queries) and a cache. During query execution, the engine interacts with the UPnQ file system to get data from physical files via wrappers, potentially using the cache to improve response time.

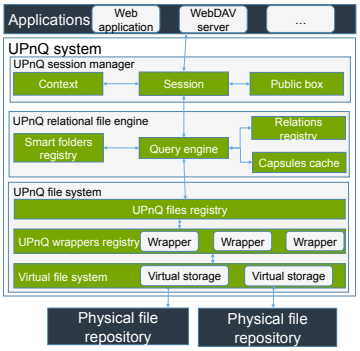


Fig. 3. Architecture of the UPnQ system

4 Experiments on the Query Execution Strategies

We conducted a simulation study to comparatively assess the performance of four different query processing, measuring query execution time, query response time and cumulative CPU usage (unit is an abstract *symbolic instant* (si)). The four assessed strategies are *Eager*, *Lazy*, *UPnQ* and *A-UPnQ* (for *Active UPnQ*). Eager is the typical case of PIM where every file is preprocessed and stored in a database, leading to a high bootstrap cost. Lazy is the opposed case where data is only fetched when needed, leading to a long response time. UPnQ adds a capsule-level caching method to the lazy strategy, improving the response time for queries exhibiting some redundancies. A-UPnQ adds prefetching to UPnQ, allowing to seamlessly query files which have been preprocessed or not and achieving in the end the query response times of the eager strategy.

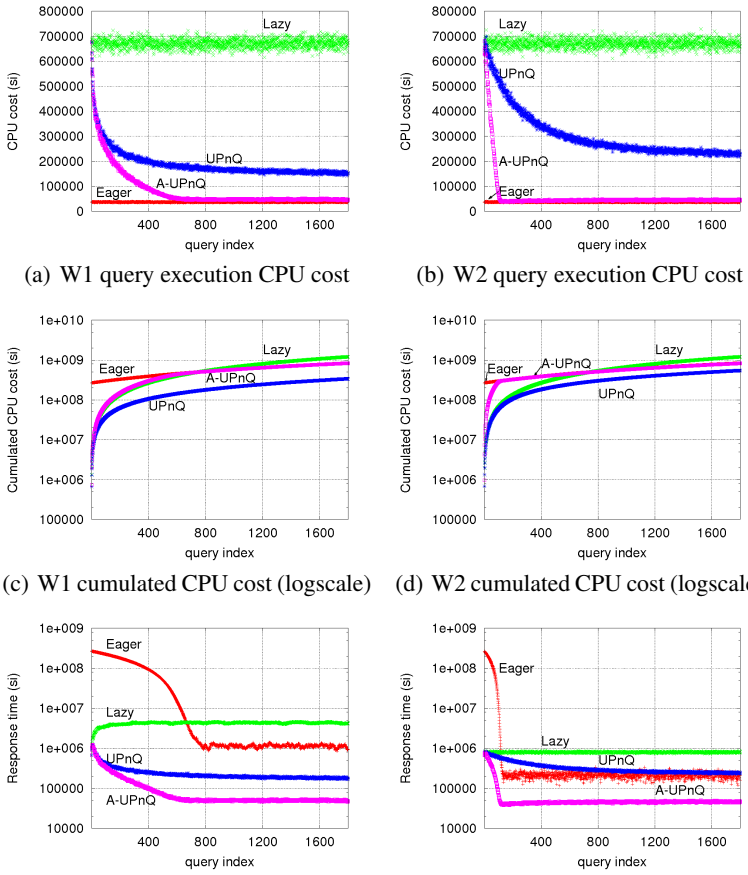
We simulate a file environment of 50000 files with 100 different file types (10 to 100 capsules). A workload is composed of 1800 queries that examines 1 to 30 capsules from 1 to 1000 files, based on two Zipf popularity curves. Queries arrive at the system following a Poisson process, interleaved with 200 random insertions of 1 to 1000 new files and 100 random updates of 1 to 100 files.

Figure 4 depicts the obtained results for 2 opposite workloads. In Workload W1, queries arrive fast at the system and are likely to overlap to a significant extent. In Workload W2, queries arrive less frequently at the system and have a small overlap. For both workloads, we simulated smart wrappers on expensive files.

As expected, the CPU cost per query (Figures 4.(a) and 4.(b)) is bounded by the Eager and the Lazy extreme strategies. For A-UPnQ, because the system has more idle time in W2, we can see the CPU cost decreases more abruptly than for W1, since the cache fills faster. For both workloads, A-UPnQ eventually catches up with eager, running queries on files that are entirely stored in the cache. The reason UPnQ decreases slower in W2, is that queries don't overlap as much as they do in W1, so cache hits are less likely. The cumulated CPU cost (Figure 4.(c) and Figure 4.(d)) shows the bootstrap disadvantage of Eager, the same for both workloads. A-UPnQ is bounded by Eager. UPnQ has the best performance in this case, since it only processes data retrieval queries and cache invalidation during updates. Query response times for both workloads (Figure 4.(e) and Figure 4.(f)) shows we have found workloads for which A-UPnQ outperforms Eager. This happens because of their different behaviors when processing inserts and updates. UPnQ tends to Eager as well in W2 (Figure 4.(f)). When queries are fast and overlap more, UPnQ surpasses Eager in an obvious manner (Figure 4.(e)).

5 Related Work

File systems are not a viable choice for personal data management, since searching files is either restricted to their name, physical location, or metadata maintained by the file system, or performed on format-dependent tags content by dedicated applications (ID3 for music files, EXIF for photos). *Personal Information Managers* (PIM) are becoming increasingly popular, but display in turn their own disadvantages. Some PIMs like PERSONAL [2] offer a homogenous management of user files but stick to a file granularity for querying. Others like EPIM [1] and MyLifeBits [8] manage more abstract



(a) W1 query execution CPU cost (b) W2 query execution CPU cost

(c) W1 cumulated CPU cost (logscale) (d) W2 cumulated CPU cost (logscale)

(e) W1 query response time (logscale) (f) W2 query response time (logscale)

Fig. 4. Simulation of workloads W1 and W2 with expensive files and smart wrappers (500 runs)

user items and their relations, and then lose the link with concrete user files. Many PIMs can indeed do a great job at easing information management as compared to file systems, but they do so on specific, delimited islands of data, and each PIM manages information in its own, personal way, usually lacking file content querying capabilities. The compartmentalised PIM support issue is raised in [7].

Finally, we mention SQLShare [11] and NoDB [4]. SQLShare is an ad hoc databases management platform which enable SQL querying over data scattered across multiple files. NoDB is big-data-oriented and maintains the main features of a DBMS without requiring data loading: *data-to-query* time is minimized up to the point where query processing directly reads raw files. Both systems however do not address issues related to file-specific model or query language (only classic SQL), and do not propose an architecture to deal with the many file types found in a user repository (only CSV files).

6 Conclusion

In the context of personal information management, we propose a new approach for managing data stored and updated in user files, namely the UPnQ principled approach, as user files are seen as the primary source of user-managed personal information. Our objective is to declaratively query the contents of dynamically discovered files at a fine-grained level. We defined a file model that allows a homogeneous representation of structured heterogeneous personal files, abstracted as UPnQ files. We introduced wrappers that extract data from files, as a core component of our architecture. We then defined a declarative file-oriented query language that allows application developers to express queries over files. Experimental results show that the UPnQ system can attain reasonable performance, and even outperformed query response time of an eager strategy for a reasonable workload. Due to lack of space, we omitted the results of our complete set of experiments, with 16 other different workloads and with different combinations of wrapping cost and granularity. Nevertheless, they are also favorable to UPnQ and A-UPnQ strategies.

Our current research direction is the integration of the UPnQ vision in the secure context of Personal Data Servers [5], which aims at giving back control over their files to users. In this setting, the severe hardware constraints imply that large amount of data (up to several Gigabytes) stored in files needs to be queried using a few Kilobytes of RAM and low CPU power. The embedded database engine should still provide acceptable performance, which entails the need for an embedded query execution model.

References

1. Essential PIM. `http://www.essentialpim.com/`.
2. Personal. `https://www.personal.com`.
3. S. Abiteboul, O. Benjelloun, and T. Milo. The Active XML project: an overview. *VLDB J.*, 17(5):1019–1040, 2008.
4. I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki. NoDB: Efficient Query Execution on Raw Data Files. In *Proceedings of SIGMOD'12*, pages 241–252, 2012.
5. T. Allard, N. Anciaux, L. Bouganim, Y. Guo, L. L. Folgoc, B. Nguyen, P. Pucheral, I. Ray, I. Ray, and S. Yin. Secure Personal Data Servers: a Vision Paper. *PVLDB*, 3(1):25–35, 2010.
6. N. Anciaux, W. Bezza, B. Nguyen, and M. Vazirgiannis. MinExp-card: limiting data collection using a smart card. In *Proceedings of EDBT'13*, pages 753–756, 2013.
7. R. Boardman. Workspaces that Work: Towards Unified Personal Information Management. In *Proceedings Volume 2 of the 16th British HCI Conference*, 2002.
8. J. Gemmell, G. Bell, and R. Lueder. MyLifeBits: a personal database for everything. *Commun. ACM*, 49(1):88–95, 2006.
9. Y. Gripay. *A Declarative Approach for Pervasive Environments: Model and Implementation*. PhD thesis, Institut National des Sciences Appliquées de Lyon, 2009.
10. Y. Gripay, F. Laforest, and J.-M. Petit. A Simple (yet Powerful) Algebra for Pervasive Environments. In *Proceedings of EDBT'10*, pages 359–370, 2010.
11. B. Howe, G. Cole, N. Khoussainova, and L. Battle. Automatic example queries for ad hoc databases. In *Proceedings of SIGMOD'11*, pages 1319–1322, 2011.
12. S. Surdu, Y. Gripay, V.-M. Scuturici, and J.-M. Petit. P-bench: Benchmarking in data-centric pervasive application development. In *Transactions on Large-Scale Data- and Knowledge-Centered Systems XI*, volume 8290 of *Lecture Notes in Computer Science*, pages 51–75. Springer Berlin Heidelberg, 2013.