Context-Aware Access to Heterogeneous Resources through on-the-fly Mashups[⋆]

Florian Daniel, Maristella Matera, Elisa Quintarelli, Letizia Tanca, and Vittorio Zaccaria

Politecnico di Milano, Dipartimento di Elettronica, Informazione e Bioingegneria
`{florian.daniel,maristella.matera,elisa.quintarelli,letizia.tanca,vittorio.zaccaria}@polimi.it`

Abstract. Current scenarios for app development are characterized by resources so rich that often overwhelm the final users, especially in mobile app usage situations. It is therefore important to define design methods that enable dynamic filtering of the pertinent resources and appropriate *tailoring* of the retrieved content. This paper presents a design framework based on the specification of the *possible contexts* deemed relevant to a given application domain and on their mapping onto an *integrated schema* of the resources underlying the app. The context and the integrated schema enable the instantiation at runtime of templates of app pages in function of the context characterizing the user's current situation of use.

Keywords: Context-aware data access, service selection, mashups, CAMUS

1 Introduction

In the last decades, the pervasive introduction of ICT technologies in our society has changed the way people access information. Traditional data management systems have left the place to sophisticated data integration systems that combine and expose rich information extracted from all kinds of sources and make it available through different media devices. Also, users have changed their attitudes and behavior and are now "digital" and "social", independently of their current usage situation and device. Yet, this flexibility does not come at a low price, as finding information that is most suitable to the users' current context may require a significant time and effort, especially if the used software does not leverage on the users' context [15].

As highlighted in [13], in order to facilitate the development of software (for any kind of device) that is able to take into account the user's context, it is important to define design methods that natively support the dynamic selection and filtering of pertinent resources and the consequent tailoring of retrieved

[⋆] This research is supported by the IT2Rail project (EU H2020 program, grant agreement no: 636078) and the Italian project SHELL (CTN01 00128 111357) data. Treating the dimensions characterizing the *context of use* as first-class design artifacts can enable context-awareness [15] in the data access layer and can guide the definition of context-aware queries over available heterogeneous resources.

To respond to the need for a development method with native support for context-awareness, this paper presents a design framework for the fast development of apps that revolves around (i) the explicit specification of the *context dimensions* deemed relevant in a given application domain and (ii) their mapping onto an *integrated schema of the available resources.* The scenarios we support have all the ingredients of data mashups or service compositions. However, differently from conventional mashup approaches, our mashups do not produce stand-alone applications, but serve as small, on-the-fly data integrations to be embedded into generic applications that require the fetching of data from heterogeneous resources. These "mini mashups" are formulated as context-agnostic queries, automatically turned into context-aware queries by the framework.

Running Example. To illustrate our method, we make use of a tourism scenario where an app personalizes the provided contents on the basis of the traveler contexts (e.g., current location and time, possible disabilities, user's preferences about topics and means of transportation). The app gathers contents about restaurants, hotels and itineraries from different resources, i.e., Web APIs and datasets that may be public or made available by the service provider who offers the app. In the scenario, local, proprietary data are, for instance, user profiles, buying histories, or similar core assets of the application to be developed. We in particular assume that data about affiliated hotels and discounts are stored in a local database table (Hotel). As for remote sources, we assume the app leverages on two external services for the calculation of itineraries (ItinSvc1 and ItinSvc2) and on one service to search for restaurants (ResSvc). As there may be multiple providers offering similar services, a service selection at runtime may be needed – again, taking into account the user's context.

Paper structure. We next introduce the concepts and artifacts of the proposed method; then, in Section 3, we go into the details of the Resource Schema, the Context Dimension Tree, and context-agnostic and -aware queries. Next we show how to interpret and execute queries and discuss an implementation in GraphQL (Section 4). Before closing we discuss related works.

2 Approach

We model context dimensions using the Context Dimension Model (CDM) [3], a specific abstract representation that, on the side of the model of the non-contextual features, provides an intuitive way to visually depict context information and the conceptual relationships that exist among the properties of context in a given scenario. The approach is in line with the idea of using two separate feature models for contextual and non-contextual requirements discussed in [13]. We propose the use of the CDM in the development of context-aware applications

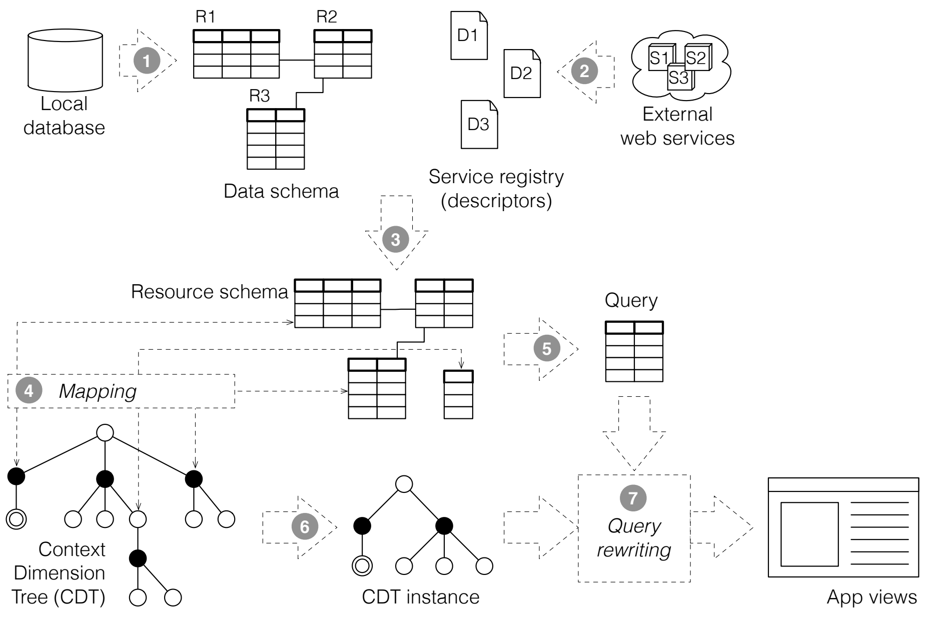

Fig. 1. Overview of approach to the development of context-aware, mobile apps starting from internal data sources, external services and a Context Dimension Tree.

to achieve two goals: (i) the concise and human- and machine-readable *representation* of all context dimensions and properties relevant to a given development scenario and (ii) the *simplification* of the development of context-aware application features. The approach turns the specification of all the possible contexts of the considered scenario into a first-class development artifact that not only serves a documentation purpose but also an operational one. The context dimensions themselves can be aggregated into categories: *information on the user* (knowledge of habits, emotional state, physiological conditions), *user's social environment* (co-location of others, social interaction, current tasks), *physical environment* (location, time, and physical conditions like noise, light, pressure, air quality). The dimensions in the latter category can be automatically derived by means of appropriate sensors, while the previous ones may require direct user input or suitable default settings.

Figure 1 illustrates the resulting method and development steps. The process starts from the identification and description of the available resources, which can be both *local data sources* ①, for example relational data bases, or third-party *Web services* represented by their *service descriptors* ②. An integrated schema of the available data is then defined, so as to present the developer with one abstract model of the data only. We call this schema the *resource schema* ③, and we express it as a *relational model.* The resource schema is generally defined manually, but it could be generated automatically depending on the regularity of the schema of the selected resources. The resource schema is accompanied by a CDT (defined in the next section) that captures all the execution contexts the application may run in ④. Each node of the CDT is mapped to the resource schema (e.g., by means of views expressed in relational algebra), in order to support automatic query rewriting at runtime.

The next step is the design of the queries that will feed the pages of the final application, e.g., the pages in a mobile app that allow the user to access and interact with the content ⑤. The resource schema focuses on the integration of data, independently of their use, and is thus context-agnostic. It is the CDT that defines which of the elements in the resource schema are related to context in the given application domain.The queries expressed over the resource schema are thus *context-agnostic* too. Neglecting context properties in this phase allows the developer to focus on the core functionality of the application, deferring adaptation concerns. The queries are typically written manually; without loss of generality, we express them in relational algebra.

At runtime, the CDT can be instantiated with concrete values coming from the *context sensors* (sensing devices, user inputs, external sources) ⑥. That is, the runtime environment of the application automatically updates the tree with context information to characterize the usage scenario the user is currently involved in. The availability of a CDT instance enables the derivation of *context-aware* queries from the context-agnostic queries, by suitably enriching them with context information ⑦. This step can be performed fully automatically, e.g., based on conventional, view-based query rewriting techniques [11].

The simplification of the development process proposed for context-aware applications therefore consists in (i) the use of a resource schema that hides technological details and data provenance issues and (ii) the automatic rewriting of context-agnostic queries defined on the resource schema into context-aware queries. Both features alleviate the developer from tasks that are typically tedious and time consuming. In the following sections we describe the core ingredients of the method. We will also show how the conceptual approach can be naturally mapped into state-of-the-art implementations making use of GraphQL.

3 Resource Schema and Context-Aware Queries

The initial activities in the development process are the *selection* of the resources of interest, which can be both local and remote data sources, and their *technical description*, specifying the details that are needed to access them. For local data sources such details refer to (i) the *endpoint* of the data source (e.g., its IP address), (ii) the *port* though which the source can be accessed, and (iii) the *username* and *password* identifying the user that represents the application. To programmatically access external Web services (e.g., SOAP/WSDL or RESTful services), it is necessary to specify: (i) the service *endpoints* (one or more URIs), (ii) the *operations* offered by the service, (iii) the respective *input parameters* and *output data schemas* and serializations (e.g., JSON or XML), and (iv) possible *authentication* details (e.g., usernames/passwords or developer keys). Data sources may differ in the communication protocols they use (plain

HTTP vs. SQL connectors), their implementation technologies (as long as they expose an HTTP or SQL interface), the data formats and schemas they use (as long as data can be correctly extracted). All these properties are specified in the respective registry entries.

For the sake of brevity, we do not further detail all technicalities here: they represent state-of-the-art development practice. What is relevant in our framework is the specification of the *access patterns* that can be used to access the services, as most services support different mandatory or optional input parameters to access data [5]. The specification of access patterns will be illustrated in Section 3.2.

3.1 Resource Schema and Context-Agnostic Queries

Once the different resources are registered in the system, it is possible to derive the resource schema of the available data so as to present the developer with a unique schema of the data. The schema aims to represent the data provided by the resources and their relationships in a way that accommodates the requirements posed by the specific application domain. For example, we can think of the tables introduced in our running example as the result of a modeling activity that produces a data schema representing all the resources selected for the given application domain.

Considering the resources identified for our running example, a possible resource schema could be the following one, where Restaurant represents the data accessible through the service ResSvc, Hotel represents the data stored in the local database HotelTab, and Itinerary represents the itineraries computed by the two services ItinSvc1 and ItinSvc2:

Restaurant(name, address, phone, type, cuisine_type, playground)
Hotel(name, address, category, childcare)
Itinerary(from, to, type, directions, price)

The identification of the previous relations depends on opportunistic choices in the app design. For example, the attribute Itinerary.`directions` is not further specified, as it is not necessary to further split and query direction descriptions. `directions` could however contain a map image, a list of instructions, e.g., about how to go from a hotel to a restaurant, or similar. The resource schema also makes only use of logical `addresses` (city, street, number), instead of physical GPS coordinates.

In addition to these application-specific design choices, the resource schema keeps track of the provenance of each attribute in the defined relations. This is achieved using an additional table: Source(id, attribute, registry_entry) that tracks the necessary, minimal meta-data: for each attribute in the resource schema, it contains a link to the registry entry that may provide data for the attribute. For the attribute Itinerary.`directions` the table will thus contain two entries, i.e., ItinSvc1 and ItinSvc2. These meta-data will be used only at query execution time and are not accessible to the developer, who instead is now able to write her context-agnostic queries. For example, the following query

$$Q_1 = \Pi_{\substack{name,address,\\ phone,type,\\ directions}} \sigma_{cousine\_type=\$VAL} \textsc{Restaurant} \bowtie_{address=to} \textsc{Intinerary}$$

is used to instantiate the page of the application that, at interaction time, allows the user to choose a type of cuisine: the user choice will replace the parameter $\$VAL$, the `from` attribute for the calculation of itineraries will be taken from the context. The app thus shows the restaurants matching the cuisine choice along with the itineraries to reach them. In Section 3.4, we show how to inject context into Q_1 to obtain its context-aware version.

3.2 Resource Mapping

Given the above resource schema, each access pattern to a resource can now be expressed as a view over it. This equips the pure technical registry entry, that tells how to interact with the service, with a semantical mapping of the service to the resource schema that enables the context-aware service selection at runtime. For instance, the chosen restaurant search service can be expressed as follows:

$$\textsc{ResSvc} \equiv \Pi_{\substack{name,address,phone,\\ type,cuisine,playground}} \sigma_{\substack{address=\boldsymbol{\$optional}\wedge\\ type=\boldsymbol{\$optional}\wedge\\ playground=\boldsymbol{\$optional}\wedge\\ cuisine\_type=\boldsymbol{\$optional}}} \textsc{Restaurant}$$

In bold we highlight two keywords that are needed to express a service's access pattern: the values `$mandatory` and `$optional` tell, respectively, if an attribute is a *mandatory* or *optional* input of the service. Both keywords are automatically replaced at runtime by their respective values. Without proper values for mandatory inputs, the service cannot be invoked; optional inputs may be used to restrict the output data produced by the service.

In addition to mandatory and optional inputs, it is also possible to specify *constant* values for some input parameters. Doing so binds the view representing the access pattern to the given value, expressing that the service is able to provide only data that complies with this restriction. If we take, for instance, two services that provide itinerary information (e.g., ItinSvc1 about a city's local transport network and ItinSvc2 about a national railway network), we may obtain the following two access patterns:

$$\textsc{ItinSvc1} \equiv \Pi_{\substack{from,to,type,\\ itinerary,price}} \sigma_{\substack{from=\boldsymbol{\$mandatory}\wedge\\ to=\boldsymbol{\$mandatory}\\ \wedge type=\boldsymbol{\$optional}}} \textsc{Itinerary}$$

$$\textsc{ItinSvc2} \equiv \Pi_{\substack{from,to,type,\\ itinerary,price}} \sigma_{\substack{from=\boldsymbol{\$mandatory}\wedge to=\boldsymbol{\$mandatory}\\ \wedge type=\text{“train”}}} \textsc{Itinerary}$$

The second access pattern explicitly binds the attribute `type` to the value `"train"` as ItinSvc2 is able to provide only data about train connections, while ItinSvc1 may provide data about all among trains, undergrounds, busses and trams within its geographical area of competence (for simplicity, we do not represent this limitation here).

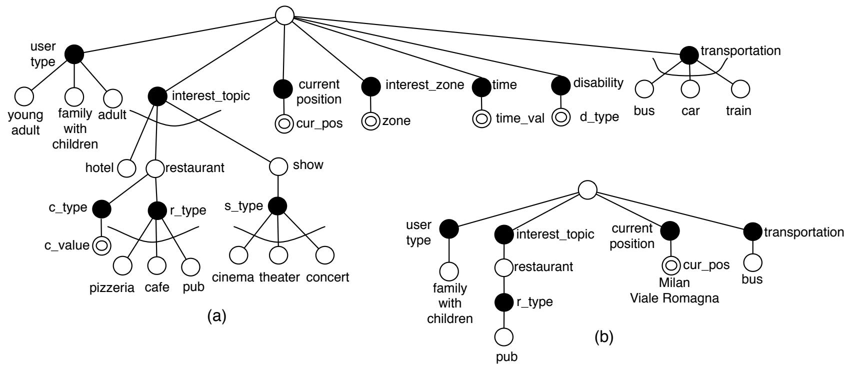

Fig. 2. The CDT of our example (a) and a context instance (b)

Analogously, the HotelTab table of the local database can be expressed as follows (observe that, for consistency with the mapping of Web services, *all* attributes are *optional* as relations do not have access patterns):

$$\text{HotelTab} \equiv \Pi_{\substack{name,address,\\ category,childcare}} \sigma_{\substack{name=\textbf{\$optional} \wedge address=\textbf{\$optional}\\ category=\textbf{\$optional} \wedge childcare=\textbf{\$optional}}} \text{Hotel}$$

3.3 Context-Aware Queries

The Context Dimension Model [3] allows one to represent Context Dimension Trees (CDTs). An example of CDT for the touristic scenario of the running example is shown in Figure 2. *Dimension nodes*, depicted in black, represent the different perspectives describing context (e.g., `user type` and `transportation`), while *concepts*, depicted as white nodes, are the admissible values of each dimension (e.g., the concepts `adult`, `young adult` and `family with children` are values for the `user type` dimension). *Attributes*, represented by double circles, are parameters whose values are dynamically derived from the environment or provided by the users themselves at execution time, and used to replace a high number of concepts when it is impractical to list them all: e.g., the `current position` dimension has as child the attribute `curr_pos`.

A context instance is a subtree of a CDT (also represented as a set of `<dimension=value>` pairs) where the parameter nodes are replaced with concrete values. Figure 2(b) shows graphically the instance C = {`user type = family with children`, `r_type = pub`, `current position = Milan Viale Romagna`, `transportation = bus`}: a family with children, currently located in Viale Romagna in Milan, moves around by bus and prefers to eat at pubs.

3.4 Query Rewriting

Following an approach similar to [3], we propose that the designer associates each *context element* (i.e., `<dimension=value>`) of the CDT with one or more

Context Element	Relational Algebra Expressions
`user type = family with children`	$\sigma_{playground=\text{“}yes\text{''}}$ RESTAURANT $\sigma_{childcare=\text{“}yes\text{''}}$ HOTEL
`r_type= pub`	$\sigma_{type=\text{“}pub\text{''}}$ RESTAURANT
`current position = Milan Viale Romagna`	$\sigma_{from=\text{“}Milan\ Viale\ Romagna\text{''} \wedge to=\$Value}$ ITINERARY $\sigma_{address=\text{“}Milan\ Viale\ Romagna\text{''}}$ RESTAURANT $\sigma_{address=\text{“}Milan\ Viale\ Romagna\text{''}}$ HOTEL
`transportation = bus`	$\sigma_{type=\text{“}bus\text{''}}$ ITINERARY

Table 1. Relational Algebra Expressions associated with the context instance C.

$$\begin{tikzcd}
& \Pi_{\text{NAME,ADDRESS,PHONE,DIRECTIONS}} \arrow[d] & \\
& \bowtie_{\text{ADDRESS=TO}} \arrow[dl] \arrow[dr] & \\
\sigma_{P1} \arrow[d] & & \sigma_{P2} \arrow[d] \\
\text{RESTAURANT} & & \text{ITINERARY}
\end{tikzcd}$$

Fig. 3. Query Q_2 where P_1 : $cuisineType$ = "*indian*" $\wedge$ r_type = "*pub*" $\wedge$ $playground$ = "*yes*" $\wedge$ $address$ = "*Milan Viale Romagna*" and P_2 : $type$ = "*bus*" $\wedge$ $from$ = "*Milan Viale Romagna*".

relations of the resource schema, filtered on the basis of the `value` of the context element itself. For instance, Table 1 shows the expressions associated with the context elements in C.

Suppose now that at run time, while in context C, the app prompts the user with the context-agnostic query Q_1 from Section 3.1, where restaurants can be selected on the basis of a cuisine type chosen by the user. Q_1 will be expanded with the user's choice and automatically rewritten using the context-aware expressions in Table 1. The result is the context-aware query Q_2 represented in Figure 3 as a standard syntax tree for relational algebra.

Note how the agnostic query is extended by adding conditions related to the context dimensions (e.g., the user's current position or the type of user) to the selection and join operations: in Table 1, the expression $\sigma_{from=\text{“}Milan\ Viale\ Romagna\text{”} \wedge to=\$Value}$ ITINERARY, associated with the context element `current position = Milan Viale Romagna`, contains a parameter referring to the destination of the itinerary, that takes now the value specified by the join condition of the agnostic query Q_1.

4 Query Interpretation and Execution

In this section we describe the synthesis of a query execution plan from the context-aware query and the consequent selection of the related data and services; in the last subsection we then give an account of the real execution flow of our framework, as it has been made concrete in a research prototype [8].

4.1 Query Interpretation and Service Selection

Conceptually, the generation of the query execution plan proceeds as follows:

1. We produce a tree-like representation of the relational algebra query that is extended with an explicit representation of the corresponding predicates.
2. We map a service to one or more query primitives in the original tree by exploiting the structure of their access pattern represented as a *tile pattern*[1].
3. We visit the mapped tree to produce a schedule of web services invocations and database assesses.

For example, let us consider query Q_2 of Figure 3. To see how this query can be mapped to an actual query execution plan involving service requests, we consider the services described in Section 3.2. Every service exposes a potential set of filtering predicates to be used when accessing it. To capture this information, the tile pattern covers not only the abstract relational algebra operation associated with the service, but also the syntactic structure of valid predicates. Here and in the rest of this section we extend with a double-line arrow the representation of the associated logical predicate. For example, Figure 4(a) shows the subtree pattern in the original query the ItinSvc1 service can answer to (section 3.2). The pattern specifies that such a service can be selected when part of the original query aims to select itineraries where:

- the `type` attribute is optional (as represented by the parentheses),
- the `from` attribute is mandatory,
- the `to` attribute is mandatory.

Figure 4(b) shows, similarly, the pattern associated with ResSvc. ItinSvc2 pattern is not shown, as its restriction `"type=train"` is not compatible with the query's request for a bus (`"type=bus"`).

We now can expand query Q_2 by exposing the logical structure of its predicates. We then note that the only registered access patterns that allow valid matches are ResSvc and ItinSvc1 because ItinSvc2 cannot pattern match. Figure 5 shows the result of the match operation; since we deal with a logically correlated sub-query, we mark with a small circle each tile pattern leaf that corresponds to an attribute involved in a join condition appearing above it in the tree.

After having identified the services by pattern matching, we schedule the service invocation by traversing the tree in post-order and executing joins with a nested join strategy. In particular, we use the convention that the left relation is always the outer relation and exploit the commutativity of the join operator to reorder the nodes in the tree and optimize the query execution. In particular, we order nodes from left to right by ascending number of correlated attributes.

[1] A tile pattern is a tree template with one or more wildcards that can match any subtree of the original query. Note that, in this view, data coming from lower nodes is an "input" to a service, while the root of the node is the "output" of the service.

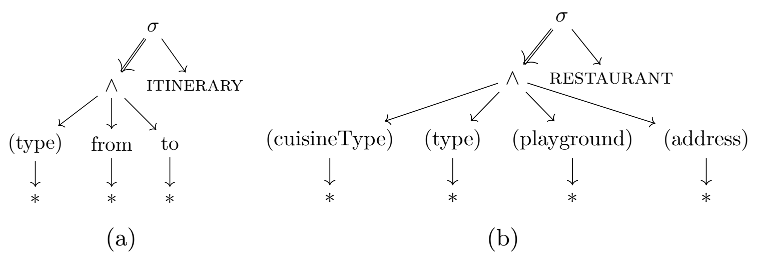

Fig. 4. Itinerary service tile (ItinSvc1) and restaurant service tile (ResSvc).

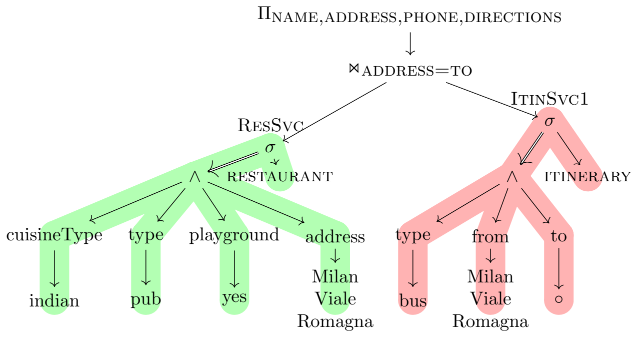

Fig. 5. Example context aware query where predicates have been expanded and services have been verified to match (see annotation).

In this case, ResSvc is invoked first (using as parameters the values of the leafs that allowed the match), while ItinSvc1 is invoked for each tuple returned by ResSvc by using the "address" attribute value as the "to" parameter value.

More refined methods to enlarge the space of solutions considered for tile pattern matching can be also adopted [5]. For example, to address the case when multiple valid services can serve a specific query, we already experimented techniques (i) to invoke them all and then fuse their outputs and also (ii) to apply some ranking strategy, for example based on service quality criteria, and then select the best service [6, 8].

4.2 Prototype implementation

As a concrete implementation of the conceptual approach discussed in the previous sections, we developed a prototype based on node.js. The prototype implements a GraphQL server-side runtime [1] that supports the execution of queries over a GraphQL schema. The schema consists of simple type declarations and describes the data to be exposed to the front-end applications, independently of any specific database, storage engine or service access logic. This feature natively supports the implementation of our (virtual) resource schema inside a GraphQL API and the proxying of incoming queries to the respective resources.

The architecture of the prototype is shown in Figure 6. To describe its main functions, let us consider a mobile app whose task is to render the example query Q_1 as defined in Section 3.1. The process for query execution starts with

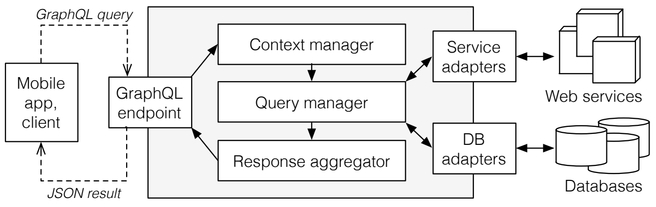

Fig. 6. Functional architecture of the prototype implementation.

the mobile app sending a GraphQL query with both the query parameters and the context data. Then the following actions are taken:

– A *Context Manager*, given an instance of the CDT and possible user inputs, decides if/which user inputs overwrite which context parameter.
– A *Query Manager* selects and invokes the corresponding data sources according to the techniques presented above (leveraging on suitable service/DB adapters for the communication with the sources).
– A *Response Aggregator* composes the integrated result set into a JSON structure and sends it back to the mobile app.

Given the internal logic of the GraphQL API and the nature of the CDT, no complex, structural query rewriting is needed: it suffices to ship the set of context properties as parameters along with the original query from the client to the API. Depending on the presence or not of context properties in the CDT, the API can then internally apply suitable selection conditions on retrieved results. As we show next, this means that the effect of adding context essentially translates into enriched selection conditions.

To exemplify the internal logic of the API, let's consider the GraphQL schema in Figure 7. `Restaurant` and `Itinerary` are the two relations of the resource schema, `CamusContext` is the CDT structure (limited to the properties of the example) and `restaurant` is the type of query supported by the API. Note how the attribute `reachThrough` of `Restaurant` supports the calculation of the join of Q2. Also note how this join must be specified at schema level (and then supported by the internal API implementation), as the schema describes the structure of the output data, rather than that of the underlying data.

Figure 8 now shows the GraphQL query corresponding to Q_2 as sent from the client to the API. The query asks for restaurants, provides the selection conditions between parentheses, and then lists the properties (projection) it wants to extract about the restaurants. The presence of the `reachThrough` property among these properties corresponds to the join to be calculated between `Restaurant` and `Itinerary`. Instead of flatting out the list of attributes, the query keeps all context properties grouped as one `cdt` element (for separation of concerns); conceptually, they all correspond to possible selection conditions.

In response to this query, the GraphQL API enacts the resolver illustrated in Figure 9. First, the existence of possible user inputs that would overwrite

```
type Restaurant {                      input CamusContext {
    name:         String                 userType:        String
    address:      String                 rType:           String
    phone:        String                 currentPosition: String
    type:         String                 transportation:  String
    cuisine_type: String               }
    playground:   Boolean
    reachThrough: [Itinerary]
}
                                       type Query {
type Itinerary {
  from:       String                     restaurant (cuisine_type: String,
  to:         String                                from: String,
  type:       String                                cdt: CamusContext!):
  directions: String                                [Restaurant]
  price:      Float                    }
}
```

Fig. 7. Resource schema of the example scenario in GraphQL schema language.

```
{
  restaurant (cuisine_type : "indian",
     cdt: {
       userType: "family with children",
       rType: "Pub",
       currentPosition: "Milan Viale Romagna",
       transportation: "bus"
     })
     {
       name
       address
       phone
       reachThrough {
        type
        directions
} } }
```

Fig. 8. Example GraphQL query (Q2) to retrieve restaurants and respective itineraries.

context properties is checked (in the example, if an explicit `from` information is provided, this would overwrite the context's `currentPosition` property). Then, the Query Manager is invoked with the entities to be retrieved (ordered list), the selection conditions, and the context of the query. Internally, the Query Manager proceeds as described in the previous section in order to produce the set of requested restaurants (including the respective itineraries) as output. Finally, for each retrieved restaurant, the resolver leverages on a so-called data class (`Restaurant`), which implements GraphQL's projection logic.

5 Comparison with other Work

Different works on mobile app design describe ad-hoc solutions for the development of context-aware applications [16] in which it is difficult to identify reusable abstractions. In [9] the authors follow a more systematic approach, showing how context-aware mobile apps can be built by mashing-up components managing the app logic with reusable *context components* dedicated to capturing context events and activating related operations in the app. The approach does not provide abstractions for context modeling: the designer is in charge of configuring the context components (which basically manage user location and time) by means

```
var root = {

  restaurant: function ({cuisine_type, from, cdt}) {

    if (from) cdt.currentPosition = from; // simplified Context Manager

    var restaurants = QueryManager.get({ entities: ["restaurants", "itineraries"],
                                        cuisine_type: cuisine_type,
                                        context: cdt });

    return restaurants.map (function(restaurant) { // simplified Response Aggregator
      return new Restaurant(restaurant, cdt);
    });
 }}
```

Fig. 9. Example implementation of the GraphQL resolver answering Q2.

of parameter settings. However, the work shows how to achieve context-aware applications by means of a lightweight integration of heterogeneous and reusable components. Our approach also exploits mashup techniques. Our context-aware queries can in fact be considered "mini data mashups" integrating on the fly selected data sources. The goal, however, is to promote the adoption of a conceptual layer (i.e., the combined use of the resource schema and the context model), which enables app developers to reason at a high level of abstraction. The adopted conceptual models then drive the automatic selection of services and their dynamic, context-aware querying at runtime.

Some other approaches offer systematic methodologies and design environments.*MoWA* [4] introduce *augmentation*, which consists in adding some scripts on top of context-agnostic pages so that at runtime context can be gathered and processed to trigger page adaptations. In line with our approach, *MoWA* promotes separation of concerns and gives context a first-class role; however, it forces the designer to add a number of scripts for each page to be adapted dynamically at runtime. The advantage of our approach is that context-awareness is achieved at the only cost of defining, during the initial design phases, an adequate conceptual model capturing the most salient context dimensions.

Further works focus on the retrieval of content from heterogeneous services. *MyService* [12] provides expert designers with the possibility to select pre-defined context-based rules on top of a service directory. Based on the chosen rules, proper services are selected at runtime depending on the gathered context, and the code of the final app invoking these services is dynamically generated. This work is in line with our idea to filter services at runtime on the basis of the identified context. However the adopted notion of context is limited to the user location, while CDM is generic enough to cover several other dimensions that, in each given scenario, might characterize the contexts of use. Moreover, in our approach the designers are not required to care about which services have to be invoked at runtime; the system selects those services that best match the identified context instance.

Some other works are characterized by the adoption of context models to guide the access to heterogeneous resources. In [7], the authors use CDM to model the possible contexts and build a platform serving the execution of a context-aware tourism app. The app flexibly collects non-structured data from varying heterogeneous sources, and provides contextual recommendations to the user. This work shows the feasibility of adopting CDM to drive the context-aware selection of services to be invoked at runtime. In addition to this, in this paper we clarify how to select services in an automatic manner and how to build related context-aware queries on top of the selected data sources.

In [2], the authors then present a Model-Driven Engineering (MDE) approach where context meta-models and model-to-code transformations guide the automatic generation of code for the final context-aware apps. The proposed techniques for model-to-code transformations are interesting and are also in line with the goal of our research. However, once meta-models are in place, context modeling for the generation of a specific app requires the designer to define rules (i.e., OCL expressions) specifying the context-aware behaviors to be shown at runtime. Our approach, instead, does not need additional specifications on top of the context model; strategies for service and content filtering are shaped up at runtime, depending on the way the captured context guides the rewriting of context-agnostic queries.

From an application perspective, the proposed method focuses on the context-aware filtering of data integrated from multiple sources. It does not provide for personalized recommendations of data items, a problem that is typically addressed in scenarios similar to our tourism example [10]. Recommending suitable items, once contextual data are fetched, is an orthogonal design issue that we already addressed in our previous work [14].

6 Conclusions and Future Work

The contribution of this paper is a principled definition of the design method underlying CAMUS (Context-Aware Mobile mashUpS) [8], a research project that aims at the conception of high-level abstractions for efficient data and service integration in context-aware mobile applications. The method is general in nature and can be applied in the development of any kind of information system that requires on-the-fly, context-aware data access capabilities (e.g., the method can be used to provide context-aware access to external resources if it is wrapped by a suitable API called from within an application, or it can be used to provide context-aware access to internal data sources in parallel to existing data access channels). The paper specifically focuses on the data preparation and retrieval tasks and shows how to enable the automatic rewriting of context-agnostic queries into context-aware queries by explicitly modeling what is considered context in a given application scenario. The method also provides the conceptual and technological foundation for principled context management, effectively assisting the work of the developer. The prototype implementation shows how GraphQL naturally lends itself as candidate technology for the seamless integration of the (virtual) resource schema with concrete data access logics.

We preliminarily measured the performance of service selection and invocation; results are encouraging and time log-normally distributed [8]. Next, we will generalize our prototype implementation, parameterize it, and devise suitable transformation logics able to transform the resource schema (already expressed in GraphQL schema language), the CDT and the set of context-agnostic queries into full-fledged GraphQL APIs. We also would like to look into contextual data display, context-driven discovery of services from large repositories, and visual design environments for modeling context and designing resource schemas.

References

1. GraphQL. Draft RFC Specification, Facebook, `https://facebook.github.io/graphql`, 2015.
2. A. Achilleos, K. Yang, and N. Georgalas. Context modelling and a context-aware framework for pervasive service creation: A model-driven approach. *Pervasive and Mobile Computing*, 6(2):281–296, 2010.
3. C. Bolchini, E. Quintarelli, and L. Tanca. Carve: Context-aware automatic view definition over relational databases. *Information Systems*, 38(1):45–67, 2013.
4. G. A. Bosetti, S. Firmenich, S. E. Gordillo, and G. Rossi. An approach for building mobile web applications through web augmentation. *J. Web Eng.*, 16(1&2):75–102, 2017.
5. D. Braga, S. Ceri, F. Daniel, and D. Martinenghi. Optimization of multi-domain queries on the web. *PVLDB*, 1(1):562–573, 2008.
6. C. Cappiello, M. Matera, and M. Picozzi. A UI-Centric Approach for the End-User Development of Multi-device Mashups. *TWEB*, 9(3):11, 2015.
7. M. Casillo, F. Colace, M. D. Santo, S. Lemma, and M. Lombardi. A context-aware mobile solution for assisting tourists in a smart environment. In *HICSS 2017*. AIS Electronic Library (AISeL), 2017.
8. V. Cassani, S. Gianelli, M. Matera, R. Medana, E. Quintarelli, L. Tanca, and V. Zaccaria. On the role of context in the design of mobile mashups. In *RMC 2016*, pages 108–128. Springer, 2016.
9. F. Daniel and M. Matera. Mashing up context-aware web applications: A component-based development approach. In *WISE 2008*, volume 5175 of *LNCS*, pages 250–263. Springer, 2008.
10. O. Daramola, M. Adigun, and C. Ayo. Building an ontology-based framework for tourism recommendation services. *Information and communication technologies in tourism 2009*, pages 135–147, 2009.
11. A. Y. Halevy. Answering queries using views: A survey. *VLDB J.*, 10(4):270–294, 2001.
12. E. Lee and H.-J. Joo. Developing lightweight context-aware service mashup applications. In *ICACT 2013*, pages 1060–1064, Jan 2013.
13. K. Mens, R. Capilla, H. Hartmann, and T. Kropf. Modeling and managing context-aware systems variability. *IEEE Software*, 34(6):58–63, 2017.
14. A. Miele, E. Quintarelli, E. Rabosio, and L. Tanca. Adapt: Automatic data personalization based on contextual preferences. In *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, pages 1234–1237, 2014.
15. D. Salber, A. K. Dey, and G. D. Abowd. The context toolkit: Aiding the development of context-enabled applications. In *CHI 1999*, pages 434–441, 1999.
16. R. Schaller. Mobile tourist guides: Bridging the gap between automation and users retaining control of their itineraries. In *Proceedings of the 5th Information Interaction in Context Symposium*, IIiX '14, pages 320–323, 2014.