Universität Stuttgart

A Mobile Data Management Architecture for Interoperability of Resource and Context Data

Von der Fakultät für Informatik, Elektrotechnik und Informationstechnik der Universität Stuttgart zur Erlangung der Würde eines Doktors der Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von

Andreas Markus Brodt

aus Gaildorf

Hauptberichter: Prof. Dr.-Ing. habil. Bernhard Mitschang

Mitberichter: Prof. Dr. Albrecht Schmidt

Tag der mündlichen Prüfung: 11. April 2013

Institut für Parallele und Verteilte Systeme (IPVS)

2013

Acknowledgements

First of all, I want to thank my doctoral advisor, Prof. Bernhard Mitschang, for giving me the opportunity to work on this challenging topic in his research group. I would also like to thank him for his guidance, his support, and many interesting discussions over all the years. Through his guidance I learned a lot about conducting scientific research and his ideas gave me new insights into my work.

Furthermore, I want to thank my current or former colleagues at the Institute of Parallel and Distributed Systems (IPVS) at Universität Stuttgart, namely (in alphabetical order) Nazario Cipriani, Frank Dürr, Matthias Großmann, Carlos Lübbe, Daniela Nicklas, Florian Niedermann, Oliver Schiller, Holger Schwarz, and Christoph Stach. Thanks for the great time working together with you all!

Special thanks goes to Sailesh Sathish from Nokia Research Center, who brought up brilliant ideas and with whom we published several papers. Also Kate Alhola, Pertti Huuskonen, Harri Kiviahde, Olli Pettay, Josh Soref, Aarne Taube, Esko Törmäkangas, Jari Tenhunen, and Seppo Yliklaavu from Nokia and Nokia Research Center deserve to be mentioned here for supporting my work on this thesis.

Also I would like to acknowledge the great work of my students and student assistants (in alphabetical order, again): Björn Dick, Alexander Martin, Oleg Marin, Victor Miyai, Dominik Morar, Bruno Nunes, Bastian Reitschuster, Tim Waizenegger, Alexander Wobser, and Thomas Würfel. Their hard work was a great support!

Last but definitely not least, I want to express my sincere thanks to my wife, my daughter, and my parents for their continuous support, encouragement and patience during my work on this thesis.

Andreas Brodt
Gerlingen, 19. April 2013

Contents

List of Acronyms

We use the following acronyms throughout this document:

Ajax Asynchronous JavaScript and XML

API Application Programming Interface

ARM Advanced RISC Machines

ASR Area Service Register

AWM Augmented World Model

AWML Augmented World Modeling Language

AWQL Augmented World Query Language

CAN Contend Addressable Network

CPU Central Processing Unit

CSS Cascading Style Sheets

DBMS Data Base Management System

DCCI Delivery Context: Client Interfaces

DDR Double Data Rate

DMS Data Management System

DNS Domain Name System

DOM Document Object Model

eMMC embedded Multi Media Card

GML Geography Markup Language

GPS Global Positioning System

HTML Hypertext Markup Language

HTTP Hypertext Transfer Protocol

HTTPS HTTP Secure

JSON JavaScript Object Notation

LDAP Lightweight Directory Access Protocol

MMOG Massively Multiplayer Online Game

MVC Model-View-Controller

NAND Negated And

OMAP Open Multimedia Application Platform

OSM OpenStreetMap

PIM Personal Information Management

POI Point of Interest

RDBMS Relational Data Base Management System

RDF Resource Description Framework

RDFS RDF Schema

RDF-3X RDF Triple Express

REST Representational State Transfer

RFCOMM Radio Frequency Communication

RPM Rotations per Minute

RSS Really Simple Syndication

SATA Serial Advanced Technology Attachment

SDP Service Discovery Protocol

SMS Short Message Service

SoC System-on-a-chip

SPARQL SPARQL Protocol and RDF Query Language

SQL Structured Query Language

SSL Secure Socket Layer

STEAMS Spontaneous Team Meeting Solution

UAProf User Agent Profile

URI Uniform Resource Locator

URI Uniform Resource Identifier

W3C World Wide Web Consortium

WKB Well-know Binary

WKT Well-know Text

WWW World Wide Web

WAP Wireless Application Protocol

XML Extensible Markup Language

XPCOM Cross Platform Component Object Model

ZUSAMMENFASSUNG[1]

Mobile Geräte wie Smartphones, PDAs, Internet Tablets, etc. haben sich zu Allzweckgeräten entwickelt, die mit Sensoren und nahezu ununterbrochenem Internetzugang ausgestattet ihren Benutzer fast überall begleiten. Somit verwalten sie verschiedenste Daten aus dem Leben und Kontext des Benutzers. Es gibt erhebliche Überschneidungen in diesen Daten, da unterschiedliche Anwendungen ähnliche oder überlappende Datendomänen verarbeiten, z. B. e-Mail und SMS-Nachrichten, Kontakte, Multimediainhalte, Kalender, Ortsinformationen, Reisedaten usw. Nicht selten halten die Anwendungen solche Daten in getrennten Datensilos, so dass andere Anwendungen kaum oder gar keinen Zugang dazu bekommen. Der Datenaustausch zwischen verschiedenen Geräten ist höchstens auf Applikationsebene möglich. Web-Anwendungen, die heute große Mengen persönlicher Daten verwalten, haben oft überhaupt keine Möglichkeit, die Daten mit anderen Anwendungen auszutauschen. Diese fehlende Interoperabilität erzeugt Redundanz in den Daten und wirkt sich letztlich negativ auf die Benutzbarkeit mobiler Geräte aus, da das Potenzial der Daten nicht ausgeschöpft wird.

Diese Arbeit präsentiert eine Datenverwaltungsarchitektur für mobile Geräte, die Interoperabilität zwischen lokalen Anwendungen, mit anderen Geräten, sowie mit Web-Anwendungen auf der Datenverwaltungsebene ermöglicht. Hierzu wird ein zentrales Repository auf mobilen Geräten vorgeschlagen, das es Anwendungen gestattet, Ressourcen- und Kontextdaten in einem integrierten erweiterbaren Datenmodell zu verwalten und auszutauschen. Das Datenmodell basiert auf Semantic Web-Technologien und unterstützt ortsbasierte Daten. Mittels spontaner Drahtloskommunikation können die Daten mit anderen Geräten ausgetauscht werden. Über eine Web-Browser-Schnittstelle können auch Web-Anwendungen Zugang zum Repository erhalten, sofern das allgemeine Zugriffskontrollmodell des Repositorys dies gestattet.

[1] A summary of the dissertation in German

Die zentralen Beiträge zur Forschung, die diese Arbeit leistet, sind folgende:

- Es wird eine Plattformarchitektur vorgestellt [Brodt et al., 2011b], die es lokalen Anwendungen, Geräten der nahen Umgebung und Web-Anwendungen gestattet, Ressourcen- und Kontextdaten interoperabel auf der Datenverwaltungsebene auszutauschen. Die Architektur basiert auf einem zentralen Repository, das diese Daten anwendungs- und domänenübergreifend in einem allgemeinen, flexiblen und erweiterbaren Datenmodell auf Grundlage des W3C Resource Description Frameworks (RDF) verwaltet. Ein mächtiges Zugriffskontrollmodell, das eng an das Datenmodell gekoppelt ist, steuert den Zugriff auf das Repository, so dass, trotz ihrer engen Itegration, die Daten geschützt bleiben, insbesondere im Hinblick auf Web-Anwendungen.

  Diese Architektur dient gleichsam als das große Ganze, in dem alle weiteren Beiträge dieser Arbeit einen wesentlichen Teil bilden.

- Es wird ein Ansatz vorgestellt, satzorientierte Anfragen effizient in nativen RDF-Datenbanken, sog. *Triple-Stores*, auszuführen [Brodt et al., 2011a]. In der entwickelten Plattformarchitektur verwalten viele Anwendungen ihre Daten in einer zentralen RDF-Datenbank. Somit kommen objekt- oder satzorientierte Anfragen besonders häufig vor, d. h. Anfragen, die von einer überschaubaren Menge an Ressourcen jeweils viele Attribute benötigen. Es ist daher essenziell, dass solche Anfragen besonders effizient ablaufen. Bisher erzeugen RDF Triple-Stores Anfragepläne, die pro Tripel-Muster der Anfrage einen Index-Scan und einen Join benötigen [Neumann und Weikum, 2008; Erling und Mikhailov, 2009]. Dadurch wird zwar eine hohe Flexibilität bei komplexen analytischen Anfragen erreicht, die Anfragepläne erzeugen aber bei satzorientierten Anfragen hohe Kosten. Der entwickelte Ansatz nutzt die Speicherlokalität in den bestehenden Indexen von RDF Triple-Stores aus, um mehrere Attribute einer Ressource in einem Schritt zu holen. Somit steht dem Optimierer der Datenbank ein alternativer Ausführungsplan zur Verfügung, der in vielen Fällen weniger Join-Operationen erfordert. Zusätzlich wurde eine spezielle Indexstruktur entwickelt, die für dieses Zugriffsmuster optimiert ist, so dass viele Speicherzugriffe eingespart werden können. Durch diese Maßnahmen können typische satzorientierten Anwendungsanfragen deutlich effizienter ausgeführt werden, ohne die Flexibilität der Datenbank bezüglich komplexer analytischer Anfragen zu beeinträchtigen oder ein vordefiniertes Datenbankschema zu erfordern.

- Da auf mobilen Geräten Ortsinformationen besonders relevant sind, wird eine Erweiterung von RDF Triple-Stores für geographische Daten und Anfragen

vorgestellt [Brodt et al., 2010a]. Es wird vorgeschlagen, geographische Daten als Literalwerte eines komplexen abstrakten Datentyps in RDF zu modellieren. Dadurch werden die Geodaten nicht als Tripel modelliert, und sind somit unabhängig vom verwendeten Datenschema oder Modellierungsstil.[2] Geographische Anfrageprädikate werden als Filter-Funktionen in der von der W3C standardisierten Anfragesprache SPARQL formuliert, ohne eine Spracherweiterung zu erfordern. Zur Auswertung der Geo-Prädikate stehen sowohl ein Geo-Index (R^*-Baum) als auch ein geographischer Selektionsoperator zur Verfügung.

Um es dem Anfrageoptimierer zu ermöglichen, die Kosten eines Anfrageplans abzuschätzen, wurde ein Verfahren entwickelt, das die Kardinalität einer Anfrage (d. h. eines RDF-Graphmusters) innerhalb der im geographischen Anfrageprädikat spezifizierten Region approximiert. Hierzu wird der von der Datenbank abgedeckte geographische Raum zunächst in Zellen aufgeteilt. Anschließend werden bis zu einer maximalen Tiefe d alle zyklenfreien Pfade ermittelt, die innerhalb einer Zelle beginnen, wobei nur die Prädikate[3] der RDF-Tripel berücksichtigt werden. Daraus wird für jede Zelle eine sog. *Pfadbündelstatistik* erstellt, die die Häufigkeit komplexer Teilgraphmuster innerhalb der Zelle abschätzt. Das Verfahren kommt dabei ohne die Annahme statistischer Unabhängigkeit zwischen RDF-Tripeln aus, die (nicht nur) in RDF-Datenbanken zu großen Schätzfehlern führen kann [Neumann und Moerkotte, 2011].

- Für den spontanen Datenaustausch mit Geräten in der nahen Umgebung wird das Konzept von *Ad-hoc Smart Spaces* vorgeschlagen, das in Zusammenarbeit mit Sailesh Sathish vom Nokia Research Center in Tampere (Finnland) entwickelt wurde [Brodt und Sathish, 2009]. Ein Ad-hoc Smart Space ist eine spontane, hochdynamische und sich autonom organisierende Gemeinschaft von Geräten, die mittels ad-hoc-Drahtloskommunikation untereinander Daten austauschen. Im Gegensatz zu *Smart Spaces*, wie z. B. von van Gurp et al. [2008] definiert, sind Ad-hoc Smart Spaces nicht an einen bestimmten mit Sensorik und Kommunikationsmitteln ausgestatteten Ort gekoppelt, sondern basieren allein auf den Fähigkeiten der kooperierenden Geräte, ohne jegliche Infrastruktur. Ad-hoc Smart Spaces versuchen dabei nicht, existierende Technologien, wie Sensoren, serverseitige Datensynchronisation oder infrastrukturbasierte Kommunikation zu ersetzen. Sie stehen vielmehr als Alternative zur Verfügung, wenn

[2] In RDF kann im Allgemeinen nicht von einem vordefinierten Datenbankschema ausgegangen werden. Es ist aber möglich, ein solches zu definieren, z. B. mittels RDF Schema (RDFS) [Brickley und Guha, 2004].

[3] Ein RDF-Tripel besteht aus *Subjekt, Prädikat, Objekt*.

andere Technologien nicht verfügbar sind, z. B. bei fehlendem GPS-Empfang in Innenräumen oder bei schwer einzurichtenden Zugriffsmodalitäten.

Durch das hochdynamische Wesen von Ad-hoc Smart Spaces, in denen Geräte jederzeit kommen und gehen können, ist es eine Herausforderung, Informationen über die zur Verfügung stehenden Ressourcen zu bekommen. Deshalb wird eine umfassende Evaluation von Protokollen vorgestellt, die Ressourcen in Bluetooth-basiertern Ad-hoc Smart Spaces auffinden [Brodt et al., 2010b]. Es zeigt sich dabei, dass für kleinere Gerätegruppen *Request Flooding*, d. h. das rekursive Fluten von Suchanfragen durch die Gruppe, am effizientesten ist, da der exponenzielle Aufwand dieses Verfahrens bei kleinen Teilnehmerzahlen nicht zum Tragen kommt. Für größere Gruppen bietet sich das zufällige Replizieren von Ressourceninformation an *(Random Replication)*, da sein nichtdeterministisches Verhalten das Protokoll robust gegenüber Veränderungen macht. Schließlich wird anhand zweier implementierter Beispielanwendungen das Konzept von Ad-hoc Smart Spaces demonstriert.

- Zum Zweck der Interoperabilität mit Web-Anwendungen wird eine Schnittstelle im Web-Browser spezifiziert, die es den Web-Anwendungen ermöglicht, clientseitig auf das Repository des mobilen Geräts zuzugreifen. Dies wird anhand dreier implementierter Beispielanwendungen verdeutlicht. Außerdem stellen wir eine Plattform für ortsbasierte *Mashups* vor, die lokale Daten mit serverbasierten Daten in einer einheitlichen Präsentation kombiniert [Brodt et al. 2008; Brodt und Nicklas, 2008]. Außerdem stellen wir ein weiteres ähnlich geartetes Mashup-System vor, das auf Serverseite die Daten der Nexus-Föderation [Nicklas et al., 2001] einbezieht [Brodt und Cipriani, 2009]. Abschließend werden diese technologischen Möglichkeiten im Bereich mobiler ortsbasierter Browserspiele angewandt [Brodt und Stach, 2009].

Abstract

Mobile devices have become general-purpose computers that are equipped with sensors, constantly access the internet, and almost always accompany the user. Consequently, devices manage many different kinds of data about the user's life and context. There is considerable overlap in this data, as different applications handle similar data domains. Applications often keep this data in separated data silos. Web applications, which manage large amounts of personal data, hardly share this data with other applications at all. This lack of interoperability creates redundancy and impacts usability of mobile devices. We present a data management architecture for mobile devices to support interoperability between applications, devices, and web applications at the data management level. We propose a central on-device repository for applications to share resource and context data in an integrated, extensible data model which uses semantic web technologies and supports location data. A web browser interface shares data with web applications, as controlled by a general security model.

As a contribution for the platform architecture we present an approach for efficient record-oriented queries in RDF triple stores. We achieve considerably faster query times for typical application queries. This is important, as the platform mandates all applications to manage their data in the same RDF database. Furthermore, we address the integration of spatial query processing into RDF triple stores, as lots of data on mobile devices possess spatial references. We present a data and query model, an implementation approach, and a method for dedicated cardinality estimation of spatial RDF queries. As a further contribution we propose the concept of ad-hoc smart spaces to enable spontaneous data exchange between mobile devices. We present an evaluation of resource discovery protocols in Bluetooth-based scenarios. Finally we addresses interoperability with web applications, for which we specify a web browser interface to access the repository of our architecture. We integrate server-sided data using mashups and apply our results in the area of location-based browser games.

1 Introduction

Mobile devices manage lots of different kinds of data, including multimedia files, Personal Information Management (PIM) data, device profile data, location and map data, to name a few. Also, they possess several sensors that allow them to collect data about their environment, i. e. GPS receivers, cameras, or accelerometers. Moreover, devices can easily access additional data from the web, or even from surrounding devices via wireless ad-hoc networks. Different applications access and utilize all this data in different ways and for different purposes. While one application simply manages certain resources, another application may use the same resources as context information to adapt its behavior to the current situation. This complies with Dey [2001] defining context as *"any information that can be used to characterize the situation of an entity"*. An example is a calendar application that simply manages meetings, while the telephony application reads the same information to adapt the ring tone, and the dining adviser application will only recommend restaurants that are located close enough to reach the meeting in time.

1.1 Motivation: Interoperability

A large amount of data on mobile devices relates to one or more aspects of the user's life. Consequently, the user constitutes an overall focus point for many data objects, which reoccur in different application use cases. As an example, Harry, a friend of the user, has an address book entry in the user's mobile device. He tried to call three times today. Harry also sent 12 text messages and 34 e-mails, and he has invited the user to his birthday party, which takes place in Harry's house. According to image annotations, Harry is depicted on 23 photos, and his phone is currently visible in the

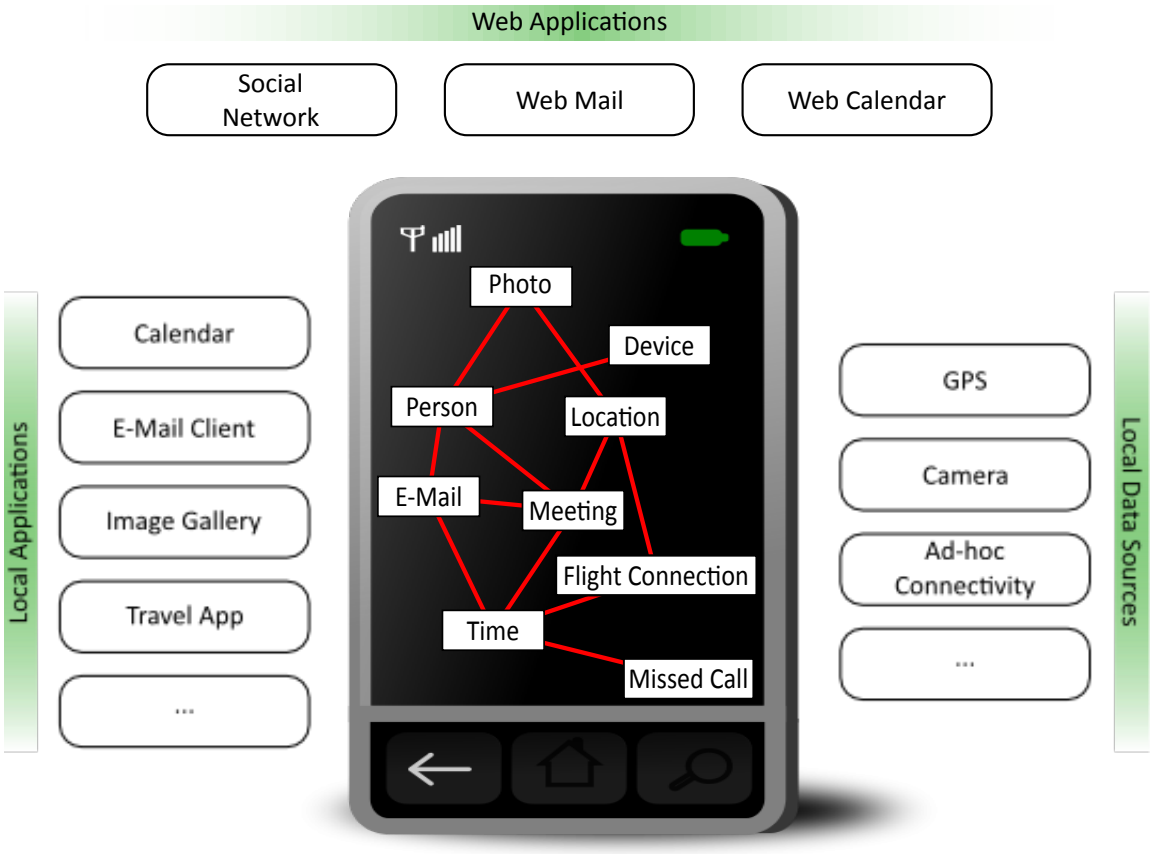


Figure 1.1: The data on mobile devices is strongly interrelated and the data domains of different applications overlap

device neighborhood. Next week, the user will fly to Glasgow with Harry to attend a conference.

As sketched in Figure 1.1, many different applications operate on overlapping data domains. In Harry's example this includes e-mail client, calendar, image gallery, travel application, address book, and many more. This is because every application deals with a certain aspect of the user's context. However, applications on a mobile device typically keep their data in isolated application-specific or domain-specific data silos. Even if such data silos are accessible via specific APIs, this causes redundant data management, software gaps, and ultimately a bad user experience. Redundant data management originates from several applications storing different aspects of the same or highly related data objects in different data silos. Software gaps are the consequence, as applications which access these objects are confronted with technical obstacles when they attempt to obtain more than one such aspect. Finally, a bad user experience follows, as the potential of the data is not exploited. What is required is *interoperability*.

1.1.1 Interoperability at the Data Management Level

Interoperability can be approached at different architectural layers of a mobile data management architecture. The most direct way of making a given application A interoperable with a fixed set of other applications S_{app}, is to exploit knowledge about how they manage their data at application level. This knowledge may be acquired from software documentation or (which is more common) through reverse engineering. This way, the application code of A may be programmed so that it utilizes the data managed by S_{app}, which directly turns A into a more useful application. The disadvantages of this approach are obvious: It makes A dependent on low-level details, such as file formats, of S_{app}. These details may change from version to version. Also, the approach is often limited to read-only access. It is not the primary focus for the application programmers of A to improve S_{app}. Thus, A benefits from the data of S_{app}, but not vice versa. Finally, if A needs to extend the data model used by S_{app}, e. g. by additional attributes, it can only do this in its own data silo, which introduces redundant data management.

A better way towards interoperability is to introduce a middleware layer that provides domain-specific APIs for important and frequently used data domains, such as contacts or messaging. As we discuss in more detail in Section 2.1.1, this approach abstracts from the low-level storage details and thus prevents applications from tinkering with files they do not own. Instead, it introduces a single data silo for all applications. However, it is restricted to a particular domain. Interoperability across different data domains, such as annotating photos with the people they depict, must be done at the application level, nevertheless.

To achieve interoperability across both applications and application domains, interoperability must start at the data management level. This means one big data repository for all applications based on a flexible and extensible data model. Only this way, resources from all kinds of domains can be interrelated without redundancies or inconsistencies. Also, only this approach enables a data management system to optimize complex queries across different domains, as it is aware of all data. Thus, in this work, we aim at a mobile data management architecture that creates interoperability at the data management layer.

1.1.2 Spatial Interoperability

The fact that mobile devices accompany the user nearly all the time adds a special aspect to the general interoperability requirement: interoperability of spatial data. In contrast to most other data management scenarios, on a mobile device a large parts of the managed data possess spatial relevance. Even entities without direct spatial

references, e. g. a phone call or a text document, may be correlated with the user's position and occur in a spatial query, such as *"where was I when I missed this phone call?"* More powerful reasoning about the user's location context is possible through spatial interoperability at the data management level, as this creates a general join criterion for most resources. Finally, spatial interoperability provides the user an additional entry point to access the data: everything that carries a spatial reference can be drawn on a map using the Point of Interest (POI) metaphor, searched by location names, or even correlated with the user's current position.

1.1.3 Interoperability between Devices

Interoperability at the data management level is required not only internally on one mobile device, but also across devices. Mobile devices possess the ability to exchange data spontaneously via wireless ad-hoc networking. Thus, data may not only overlap between different applications, but also between different devices. As an example, a meeting may be scheduled in each participant's device, and yet another device might know the geographic location of meeting room 3.436. This way, devices can work together in a similar way as humans do, e. g., when they ask people nearby what time it is or in which direction the next tram station is located. Connecting co-located devices in a spontaneous fashion results in an even better view on the user's current context, as more data is available.

1.1.4 Interoperability with Web Applications

To an increasing degree, web applications are used on mobile devices, as devices are becoming more powerful and mobile web browsers are reaching desktop class. Web applications require special attention in mobile data management scenarios, as they handle large amounts of user data. Many users describe their activities, keep track of friends, and publish digital media via web applications.

This content is a rich source of context data. Very limited interoperability is offered through proprietary service APIs which many web applications, such as social networks, shopping sites, or media sharing services, provide. Local applications may connect to the service APIs and retrieve the content of the particular application. Some mobile software platforms provide wrappers around selected service APIs, which facilitate integrating data from the particular services. Yet, interoperability again happens in application code rather than in an application-independent data management layer. It is also restricted to the services which a local application was manually programmed to support. If, for instance, the user is active in several social networks, all of them must be supported.

Moreover, if web applications would benefit from context data that is gathered locally, such as the current GPS position or data from nearby devices, they cannot receive it from servers in the cloud, but must obtain it from the client side. The strict separation between web applications and local applications makes interoperability very difficult and prevents many potential benefits. E. g., if the user books a flight online, the booking web site could guess the departure location from the user's current position and her past flights, and the destination from the calendar. Also, the web site could make the flight details appear in the user's calendar automatically—without the user needing to copy the flight data into the calendar manually.

1.2 Requirements

To achieve interoperability as introduced above, central requirements need to be addressed regarding the data model, the data management system, ad-hoc inter-device connectivity, and data access for web applications.

1.2.1 Data Model

Interoperability at the data management layer across applications, web applications, and devices, requires, first of all, a data model that is domain-independent, extensible, flexible, and that strongly supports interrelations between single data items (we will hereafter refer to them as *resources*). Extensibility and flexibility enable applications to share and augment resources by additional information, such as new attributes, context data, or so-called *tags*. Moreover, the graph nature of the resources on a mobile device, as sketched in Figure 1.1, must be well reflected in the data model. As data of different application is combined in a single model, the data model must also explicitly address access control. As a consequence of spatial interoperability, the data model must be capable of expressing spatial data and augmenting resources with spatial references.

1.2.2 Integrated Data Management System

Interoperability at the data management layer requires an integrated Data Management System (DMS) that allows for expressing the data of different applications in a single data model. The DMS must support complex analytical queries, e. g. for context reasoning across many application domains. However, it must not compromise efficiency of simple application queries for efficient analytical queries, as applications use the DMS as their primary storage back-end. In addition, the DMS must be domain-independent and capable of optimizing queries across different domains.

Moreover, the DMS must evaluate and execute the access control information that is integrated into the data model. Finally, the DMS must support integrated spatial query processing to achieve spatial interoperability.

1.2.3 Ad-hoc Inter-Device Connectivity

For interoperability between co-located devices, first of all a suitable discovery mechanism is required. Also, fault tolerance regarding frequent disconnects is necessary, as mobile devices may disappear any time. Applications which exploit data from co-located devices typically must implement the communication logic themselves in application code to solve these requirements and deal with low-level protocol details. A data management and data provisioning layer offering interoperability between devices would hide those details and offer higher-level abstractions for applications to exploit resource and context data from co-located devices easily.

1.2.4 Browser-based Data Access for Web Applications

Interoperability with web applications, naturally, mandates that web applications get access to local data on the mobile device. Partly, this requirement can be fulfilled on the server side. However, data from local sensors and co-located devices is only available on the client side. Thus, web applications require a local interface to access data on the mobile device. This can only be achieved through extending the web browser by a suitable script interface that features additional access control. Given the respective access privileges, web applications should also be able to obtain write access to the data management layer, such that other applications can again benefit from the committed data.

1.3 Contributions and Outline of this Thesis

This thesis provides a number of contributions to address the requirements of Section 1.2. We present them in the following.

- Chapter 2 presents a **platform architecture** to enable interoperability between local applications, co-located devices, and web applications at the data management level of a mobile device, as published in [Brodt et al., 2011b]. The architecture addresses all of the requirements listed in Section 1.2, most notably the general and flexible data model and the integrated data management system. Our architecture is based on a central data repository on mobile devices which all applications use cooperatively and which can be shared with co-located

devices and web applications. The data is stored and managed in an RDF-based data model and a powerful access control mechanism, which is tightly coupled with the data model, regulates access to it. This architecture serves as the big picture in which all further contributions of this thesis constitute a significant part.

- Chapter 3 presents an approach for **efficient record-oriented queries in RDF triple stores**, which we published in [Brodt et al., 2011a]. This contribution is a key enabler of an integrated data management system, as required in Section 1.2.2. Our approach exploits storage locality in the existing RDF indexes of typical state-of-the-art triple stores, and offers the query optimizer an alternative query plan that may require significantly less joins. This way, typical application queries that select a medium-sized set of resources and retrieve a fair number of attributes for each of them, may be executed significantly faster, without compromising the flexibility of triple stores to solve complex analytical queries.

- Chapter 4 addresses the **integration of spatial query processing into RDF triple stores**, as published in [Brodt et al., 2010a]. We enable RDF triple stores to process spatial queries and analyses efficiently, which is mandated in Section 1.2.2. We model geographic data in RDF as complex objects represented as literals of a complex geometry type and provide spatial query predicates in the standardized SPARQL query language without language extensions. We add both a spatial selection operator and a spatial index to an RDF triple store and provide dedicated spatial statistics for the query optimizer.

- Chapter 5 introduces the overall **concept of ad-hoc smart spaces,** which we published in [Brodt and Sathish, 2009] in cooperation with Sailesh Sathish from Nokia Research Center Tampere, Finland. Ad-hoc smart spaces address the requirement of ad-hoc inter-device connectivity stated in Section 1.2.3. As a central contribution for this, we provide a comprehensive **evaluation of resource discovery protocols** in Bluetooth-based ad-hoc smart spaces on mobile devices, which we initially presented in [Brodt et al., 2010b]. Furthermore, we present two **sample applications** to demonstrate the concept.

- Chapter 6 addresses interoperability with web applications as mandated in Section 1.2.4. We propose a **web browser interface** for web applications to access the repository of our architecture. We present a **platform for location-based mashups** combining the local GPS location (obtained via a web browser extension) with server-based location data [Brodt et al., 2008; Brodt and Nicklas, 2008] as well as a mashup system based on the **Nexus** platform [Brodt and

Cipriani, 2009]. We apply our results in the area of **location-based browser games**, as published in [Brodt and Stach, 2009].

- Chapter 7 concludes the thesis with an outlook on further research perspectives.

2

Mobile Data Management Architecture

We developed a platform architecture to enable interoperability between local applications, co-located devices, and web applications at the data management level of a mobile device, as published in [Brodt et al., 2011b]. This chapter introduces this architecture. First, we discuss the state of the art in relevant research areas in Section 2.1. We present our platform architecture in Section 2.2. The architecture is based on a central repository that manages all resource and context data in a single semantic data model, which we discuss in Section 2.4. As this approach raises security concerns, our platform provides a powerful access control mechanism that is deeply integrated into the system architecture and coupled with the data model, as explained in Section 2.5. Finally, Section 2.6 concludes the chapter.

2.1 State of the Art

The current state of the art concerning data interoperability between applications on a mobile device, is to provide domain-specific APIs. Our architecture is strongly based on Semantic Web technologies and borrows ideas from the Semantic Desktop research area. Finally, there are a number of related works for interoperability with web applications. We discuss these three topics in this section.

The state of the art and related works concerning ad-hoc inter-device connectivity do not primarily impact the design decisions of the architecture presented in this chapter, as they are orthogonal. For this reason we address them in Chapter 5.

2.1.1 Domain-specific APIs

Today, most software platforms for mobile devices provide an Application Programming Interface (API) for important and frequently used data domains, e. g. contacts or calendar. One such domain-specific API supports managing data of the same domain in the same data silo across many applications. Using a contacts API, e. g., a photo annotation app may offer a list of known persons, so that the user does not need to re-enter them. Or a public transport application may store a train connection in the calendar and create a reminder via a calendar API.

Domain-specific APIs, however, do not solve the general data interoperability problem. They are restricted to their particular domain and designed for common use cases there. Application requirements beyond that, such as introducing additional attributes, are not supported. If an application needs to store additional data, it typically must do so in its own data silo. This prevents other applications from accessing the data and reintroduces redundancy. Furthermore, not all data domains are covered by an API. The public transport application may create a calendar entry about the train connection, but without a map API, the map viewer will not be able to show the geographic location of track 6, from where the train leaves. Finally, as domain-specific APIs are nothing but front-ends for domain-specific data silos, they can only provide interoperability at application level. I. e. if an application needs to interrelate data from different domains, it has to query several APIs and combine the results in application code. Compared to a simple join in the data management layer, this is inefficient and requires more complicated application code.

2.1.2 Semantic Web and Semantic Desktop

The *Semantic Web* [Berners-Lee et al., 2001] community has developed concepts and languages, including RDF, RDFS and SPARQL, to model structured data in a way that allows creating relations between resources easily. These techniques are well-suited to augment resources with context data and make them searchable via these annotations. Semantic web technologies are generic and require a domain-specific ontology to specify resources and their relations.

The *Semantic Desktop* aimed at bringing Semantic Web technologies to the desktop and integrating applications through ontologies [Sauermann et al., 2005]. The Nepomuk project [Bernardi, 2008] developed a large software stack and ontologies [Sauermann et al., 2009; Mylka et al., 2007] for the semantic desktop. Nepomuk uses "crawlers" to search a computer and annotate files, e. g. to provide a more useful desktop search. A peer-to-peer architecture facilitates sharing files and their semantic data between users. The key ideas of the Semantic Desktop are well applicable to

our architecture and parts of the published ontologies can be utilized directly. Yet, as the semantic desktop focuses on desktop computers, there is no emphasis on context data, such as location or mobility. Also, as we outlined in [Brodt and Sathish, 2009; Brodt et al., 2010b], a mobile scenario requires wireless ad-hoc networking that is robust towards frequent connects and disconnects rather than structured peer-to-peer networks for sharing data across devices.

At Nokia Research Center, Lehikoinen et al. [2007] prototyped a framework for mobile content management. Among other things, it provided a general metadata API for all kinds of media content on mobile devices. It described the content using RDF-based ontologies. The metadata API mapped the RDF model to a generic relational model and stored it in a Relational Data Base Management System (RDBMS) on the mobile device. However, the many abstraction layers made the framework too inefficient to be used on resource-constrained mobile devices, so it never made it into production state.

2.1.3 Interoperability with Web Applications

There is currently a trend towards domain-specific APIs for web applications. Basically, they simply replicate the existing domain-specific device APIs inside the web browser as a JavaScript interface. The W3C Geolocation API [Popescu, 2009], for instance, specifies an interface for web applications to obtain the position of the user. The W3C Device APIs and Policy Working Group [Berjon et al., 2010] is in the process of defining client-side APIs to enable interaction of web applications with device *services*, such as Calendar [Tibbett and Chitturi, 2010], Contacts [Tibbett, 2010], Camera [Tran et al., 2010], etc. In the same way as domain-specific on-device APIs, these APIs are restricted to their particular domain and cover the common use cases there. As discussed above, interoperability with other domains or further application requirements are impossible.

The now discontinued W3C Delivery Context: Client Interfaces (DCCI) specification [Waters et al., 2007] attempts to standardize a generic and domain-independent context provisioning interface for web applications. It exposes a number of *properties*, which are organized hierarchically. DCCI does, however, not further specify the properties; they must be defined and standardized separately. We created an open-source implementation of DCCI [Brodt, 2007b] and utilized it in several scientific systems [Brodt et al., 2008; Brodt and Stach, 2009; Brodt and Sathish, 2009; Fenrich et al., 2009], as further discussed in Chapter 6. We also received several inquiries on our implementation from the scientific community. Nevertheless, DCCI never reached production state.

In addition, several interfaces provide client-side storage inside the web browser. Web applications often store small data items as Cookies in the web browser [Kristol and Montulli, 1997], e. g. session IDs. Gears [Google Inc., 2007] extended this principle to a complete RDBMS inside the web browser. This allows, for instance, a web mail application to store e-mails locally, so they can be read offline and only need to be downloaded once. Popular web applications, including Google Mail and Google Docs, make use of this. With the upcoming HTML5 standard several specifications were proposed to allow web applications to store data on the client: [Mehta et al., 2010] defines an indexed record store, [Hickson, 2010] drafts an SQL-based interface, [Hickson, 2011] specifies a key-value store and, at the time of writing of this document, seems the closest to standardization. Yet, these interfaces cannot provide interoperability, as they follow the same-origin policy [Ruderman, 2010], i. e., only the web application which created the data may access it.

2.1.4 Summary

The domain-specific APIs typically found in mobile software platforms provide interoperability between applications, if the same data domain is concerned. They do not provide interoperability across domains, as the data of each domain is managed in its own data silo. This prevents an integrated data model and thus interoperability at the data management level, as we required in Chapter 1.

The Semantic Web developed technologies that are well-suited for this problem; they are flexible and allow interlinking resources and augmenting them with context data. The Semantic Desktop builds on top of the Semantic Web and exploits its technologies for better interoperability and integration of data on desktop computers. This is a well related problem and some solutions and ontologies can be utilized directly on mobile devices. Yet it misses essential mobile aspects, such as mobility, spatial interoperability, or ad-hoc interaction.

The latest standardization efforts for interoperability with web applications do, in essence, recognize some of the requirements stated in Chapter 1. However, they either create domain-specific APIs, or specify strongly isolated data silos for each web application.

2.2 Platform Architecture

As depicted in Figure 2.1, our platform architecture comprises three layers: the Data Management Layer, the Data Provisioning Layer, and the Application Layer. The Data Management Layer hosts the Integrated Resource and Context Repository as the

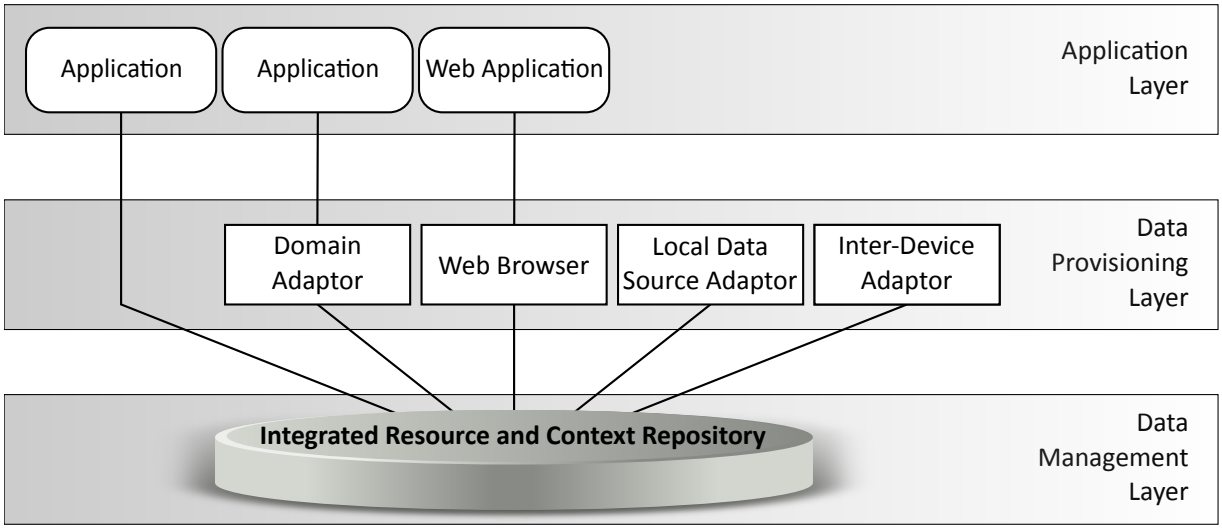


Figure 2.1: Layer architecture of our mobile data management platform

storage back-end. The Data Provisioning Layer consists of Domain Adaptors, which provide domain-specific APIs to applications that do not require data from more than one domain, local data sources (e. g. a GPS receiver or an accelerometer), the Inter-Device Adaptor managing wireless ad-hoc data-exchange with other co-located mobile devices, and, most notably, the web browser. The web browser provides a dedicated data interface for web applications. The Application Layer incorporates the applications, which ultimately consume the resource and context data (and may also contribute). The web browser interface abolishes the data separation between local applications and web-applications. Furthermore, via the Inter-Device Adaptor, applications may access data from other mobile devices transparently.

Figure 2.2 shows the system architecture of our platform in further detail. The Integrated Resource and Context Repository is the crucial component of the platform. The repository achieves interoperability between applications, co-located mobile devices, and web applications by managing their data in one central place. The data is modeled in RDF, which facilitates establishing relations between resources, as well as annotating and augmenting them with additional information. The repository supports spatial data and is able to execute spatial queries efficiently using a spatial index. In addition, it possesses a small main-memory database for dynamic data, e. g. the current GPS position. Such data is frequently updated and thus inefficient to index persistently. The repository still presents the dynamic data as part of a single consistent data model. The repository is accessed through the Data Management Interface. It accepts SPARQL queries, provided the requestor possesses the respective access rights.

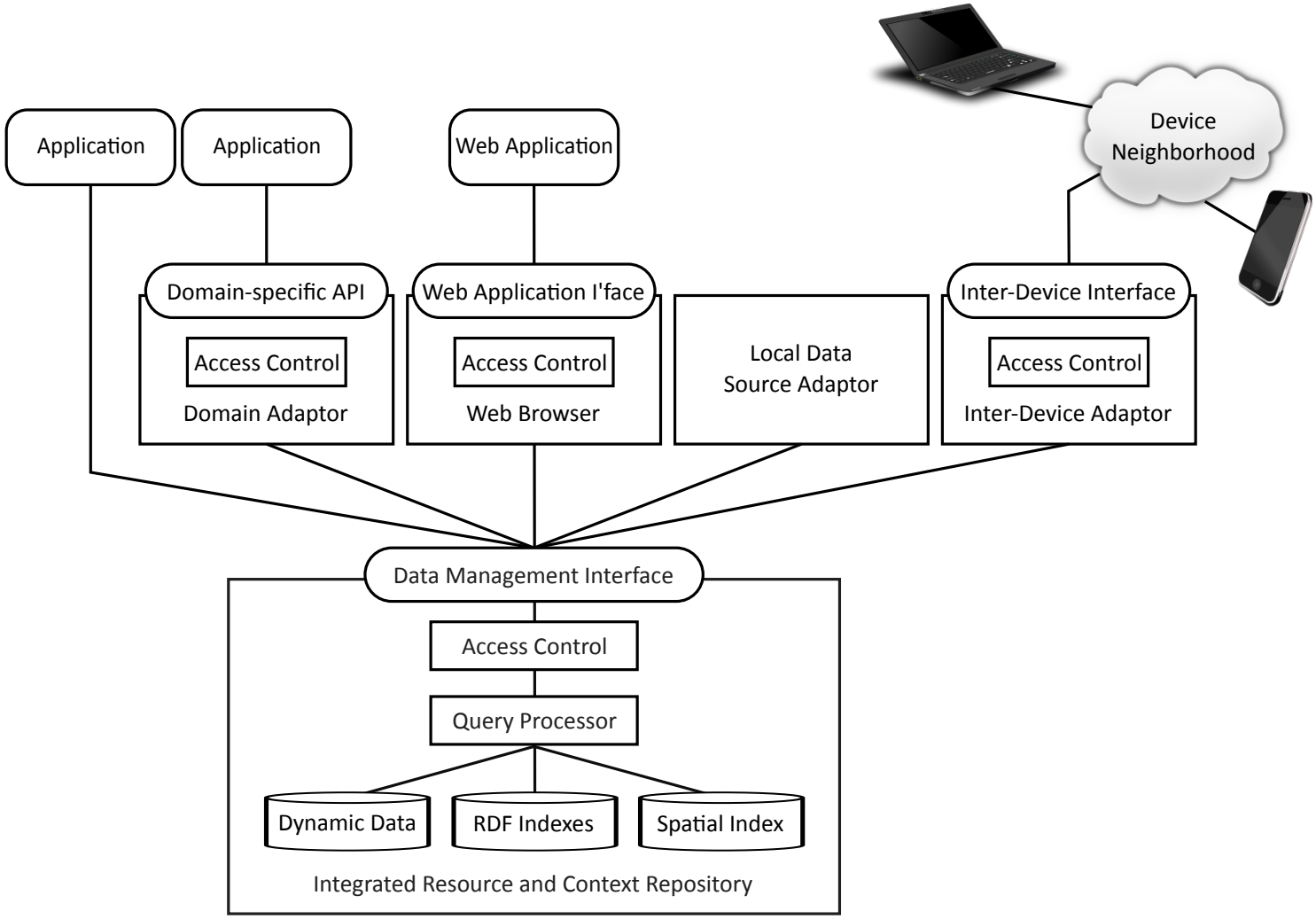


Figure 2.2: System architecture of our mobile data management platform

Context data from local data sources (sensors, etc.) is brought to the repository via dedicated adaptors. These Local Data Source Adaptors access the specific low-level API of the data source (e. g. the driver of the GPS chipset) and forward the data to the Data Management Interface. Similarly, the Inter-Device Adaptor receives data shared by co-located devices in an ad-hoc fashion and forwards it to the Data Management Interface. It also monitors the device neighborhood, performs access control for incoming requests, and forwards them to the Data Management Interface.

Installed applications can access the Data Management Interface directly to evaluate a SPARQL query. This gives them full flexibility to read and write the entire repository across all application domains, which fulfills our interoperability requirement completely. The repository performs access control checks to enforce the access rights which the application was granted.

If an application only needs data of a particular application domain, it can also use a Domain Adaptor. The Domain Adaptors offer higher-level programming abstractions through domains-specific APIs which are limited to a particular application domain, e. g. e-mail or calendar. The APIs are less flexible and do not provide interoperability across domains to applications, yet they are easier to use and can, to a certain degree, replace existing API implementations. However, unlike the domain-specific APIs discussed in Section 2.1.1, an API provided by a Domain Adaptor still caters for interoperability from other applications' point of view, as it contributes all created resources to the Integrated Resource and Context Repository. Internally, the Domain Adaptors translate their programming abstractions to calls to the Data Management Interface. Yet, as these calls are fairly restricted, domain-specific access control can be performed in the Domain Adaptors, which is much simpler and thus more efficient than access control of the Integrated Resource and Context Repository.

2.3 The Data Management Layer

The Data Management Layer is the crucial part of our mobile data management platform. It abstracts from storage and query processing details and provides an integrated view on resource and context data originating from applications, co-located devices, local data sources, and web applications. It also performs access control, effectively creating a view on the data which a particular application may access. The Data Management Layer is accessed via the Data Management Interface which passes the queries to the repository.

The Data Management Interface provides several interaction methods for applications and the Data Provisioning Layer. The most common one is the query/response interaction method. It accepts a SPARQL query string and executes it on the repository.

It returns the query result or, in case of an update, a success or error code. This interaction method satisfies the requirements of most applications. Yet for applications consuming dynamic data, most notably sensor-based context data, query/response interaction is not favorable. Therefore, the Data Management Interface also provides an event-based interaction method for dynamic data. It allows an application to subscribe to a dynamic data resource and notifies the application whenever the resource changes. This way, an application may, e. g., track the current GPS position and is notified on every position update. Finally, it is not optimal if the Local Data Source Adaptors must produce a SPARQL Update [Seaborne et al., 2008] string to indicate a change in a sensor reading, for the repository to parse. For this purpose, the Data Management Interface provides a binary interaction mode which allows supplying updates to the repository as binary data blocks. The binary interaction method may also be used for backup, bulk-loading and synchronization purposes. It bypasses the high-level query interfaces by supplying the data in its internal representation. This is more efficient, yet vulnerable to changes in the internals of the repository. Thus, the binary interaction method should only be used by software that is included in the mobile software platform.

2.4 Data Model

To achieve the desired interoperability, the central requirements for the data model are flexibility, the ability to interlink resources easily, and support for spatial data. Flexibility is required so that all applications can map their specific data model to the general one. Interoperability at the data management level comes through relations between resources of different application domains, which are impossible in isolated data silos. Thus, it must be possible to relate resources created by application *A* with other resources of which *A* is unaware. Also, other applications must be able to annotate and augment *A*'s resources with additional data. The data model must be flexible enough to support this. Finally, on a mobile device it must be possible to annotate everything with location data. The data model must support spatial data types and cater for efficient retrieval of resources by their spatial references.

We use RDF for our data model, as it is very flexible and directly supports relations as first-class objects. RDF models everything as a number of *(subject, predicate, object)*-triples which make a statement about the resource denoted by the subject. In case of an attribute of a resource, e. g. the sender of an e-mail, the predicate denotes the attribute name and the object contains the attribute value, e. g. "harry@hamster.com". In case of a relation between two resources, the subject and the object denote the resources and the predicate specifies the type of relation that connects them, e. g. to

model that “Person 117” is an attendee of “Meeting 1432”. All triples together make up one directed labeled graph. It is easy for an application to create additional triples adding further attributes or relations to a resource, which enables interoperability at the data management level. We explain RDF in more detail in Chapter 3.

To express spatial features, we use typed literals of a self-contained complex spatial data type [Brodt et al., 2010a], as further discussed in Chapter 4. Thus, we encode the spatial reference of a resource in a single RDF triple, with the object of the triple containing the spatial feature.

A further advantage of RDF is that it also covers the metadata level, i. e. the class-membership of the resources and the properties of these classes, as defined in an ontology. As the entire data model is a single graph of resources from completely different application domains, there is no structural grouping of resources as opposed to, e. g., a table in the relational data model. Thus, it is important that all resources explicitly model their class membership, so that applications can even distinguish between them. Also, pointing to the metadata level enables a resource to connect to all resources of a class, which is useful, e. g., to model access control at the data management level.

The concrete ontologies which finally express the resources are to a large degree up to the applications to define. They can easily do that by adding the triples to the repository which define the required classes on the metadata level. Yet, especially classes that are frequently used, such as PIM data, should be standardized. The mobile device software platform is likely to include vendor-specific tools, applications, and APIs, such as the domain adaptors shown in Figure 2.2, which center around a number of core classes. To achieve interoperability between devices of different vendors and device software platforms, standardization efforts are required. As stated in Section 2.1.2, ontologies from Semantic Desktop research may serve as a good starting point for this.

2.5 Access Control

As interoperability is enabled at the data management layer, access control must begin there as well. We use the principle of role-based access control [Ferraiolo and Kuhn, 1992]; applications requesting resources are assigned one or more access roles that allow certain operations on the resources. The access control model is tightly coupled to the data model and to a large degree evaluated in the Integrated Resource and Context Repository. The RDF-based data model of our platform explicitly defines all classes and their properties. This can be utilized to grant access rights to an access role on the metadata level. A role may be granted access to a class, specific properties

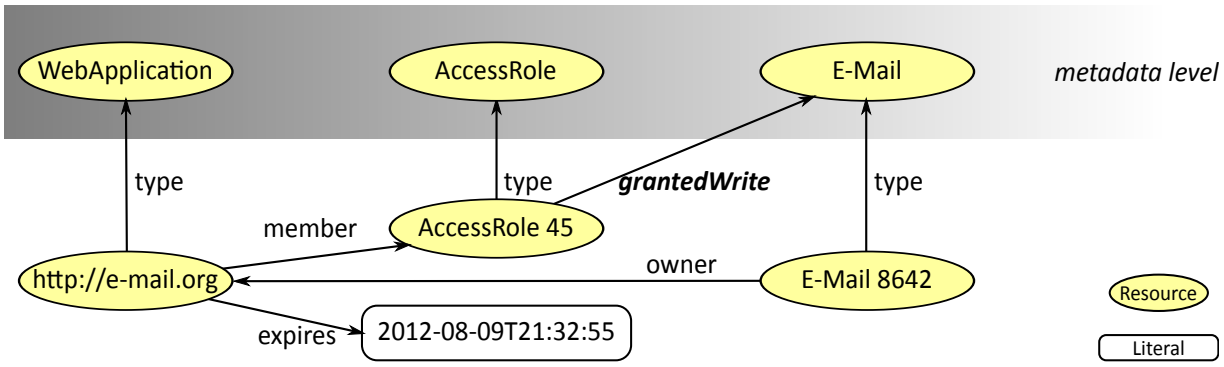


Figure 2.3: Access control based on class-membership of resources, integrated in the RDF-based data model: Members of `AccessRole 45` may write all instances of class `E-mail`

of a class, or particular resources. Finally, the owner of a resource, i. e. the application which created it, always has full access rights on the resource. Access roles may be grouped hierarchically, i. e. access roles may be members of higher level access roles. This creates the flexibility to allow very selective and fine-grained access roles and at the same time keep them manageable by a reasonably simple graphical user interface.

Figure 2.3 illustrates an access role that allows its members to write all instances of the `E-mail` class. The depicted web application `http://e-mail.org` is a member of this access role and thus allowed to write `E-mail 8642`. In addition, it is also the owner of the e-mail, so that it possesses full access rights anyway. In the example in Figure 2.4, members of `AccessRole 94` may read the `sender` property of E-mail instances, but not the content or anything else.

To gain access rights, an application asks the data management layer for one or more access roles. This triggers a graphical dialog in which the user may assign the desired access roles to the application. An installed application typically does this once at installation time. A web application may ask for access roles via calls to the Web Application Interface. The granted access roles for a web application may be remembered for a certain time period, but will eventually expire—in the same way that HTTP cookies expire. Figure 2.3 illustrates that the expiration date is stored with the web application `http://e-mail.org`. Also, if the last access role of web application expires, the entire knowledge about the web application, including its owned resources, may be removed from the Integrated Resource and Context Repository. Co-located devices may ask for access roles via the Inter-Device Interface and the decision may be remembered for a certain time.

Naturally, before an app, web application or device may obtain any access roles, it must be authenticated. To authenticate installed applications, most mobile device

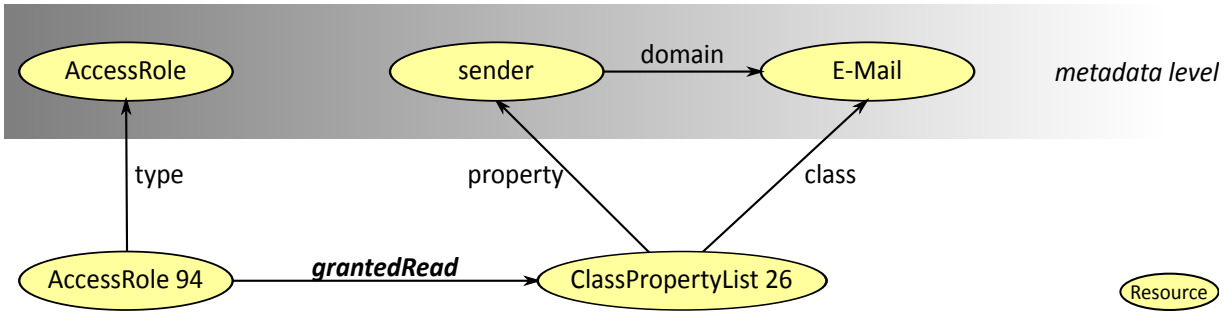


Figure 2.4: Access rights on specific properties of a class, integrated in the RDF-based data model: Members of `AccessRole 94` may only read the `sender` property of `E-mail` instances

software platforms require the application to carry a unique application ID and a digital signature [Google Inc., 2010; Forum Nokia, 2008; Apple Inc., 2010]. The application ID is used to provide platform security, e. g. to control access to communication capabilities of the device. The repository can directly use the application ID and assign access roles to it. Web applications are uniquely identified by their URI. However, to prevent man-in-the-middle attacks, web applications must use HTTPS and authenticate themselves using a trusted SSL certificate. If a web application fails to authenticate, the web browser will not accept its request for access roles. Devices connecting via the Inter-Device Interface can be identified by their network device ID.

Generally, the Integrated Resource and Context Repository evaluates the access rights of an application when a query is sent to the Data Management Interface. As the access rights are modeled together with the resource data, the query can be rewritten to restrict the query result to the resource data for which the respective access rights are present. The rewritten query checks existence of an access role which connects the application to the resource. For the most common case of access control based on class-membership, this means fetching the class of every resource and joining the class with the access roles. As most queries likely check the class of resources anyway, we consider this extra effort moderate. By contrast, access control on specific properties of a class requires more complicated rewrites which may require several extra joins with the metadata level. Thus, access control on specific properties should be used with care.

The Domain Adaptors for domain-specific APIs send ordinary queries to the Data Management Interface. Yet, they generate the queries from calls to defined programming abstractions rather than letting the calling application define the query freely. If the implementation of the Domain Adaptor is trusted—we assume it is provided by the software platform vendor—the query can be run as-is, if the application pos-

sesses the rights to access the domain-specific API. Thus, the Domain Adaptor checks whether the application is allowed to access resources of the particular domain at all. If this check succeeds, the repository does not need to perform additional access control. Similarly, the Local Data Source Adaptors are likely to be tightly coupled with the platform software and thus may circumvent access control.

2.6 Summary and Outlook

Our architecture addresses interoperability at the data management level of mobile devices. There is considerable overlap in the data which applications on a mobile device typically manage, as different applications handle similar data domains. However, a lot of this data is kept in separate data silos and is at best accessible via domain-specific APIs. The same holds for web applications, in which users keep large amounts of personal data. This lack of interoperability creates data redundancy and ultimately impacts the user experience of mobile devices.

We created a mobile data management architecture to enable interoperability between installed applications, co-located devices, as well as web applications at the data management level. Our approach is based on a central data repository on mobile devices which all applications use cooperatively. The data is stored and managed in an RDF-based data model, and powerful access control, which is tightly coupled with the data model, regulates access to it.

Future Work

Possible future extensions of this work may include a comprehensive user interface concept that fully exploits the integrated data model, as developed, e. g., by Lehikoinen et al. [2007]. Moreover, the Data Management Layer could be augmented by a full text index, as frequently found in Semantic Desktop systems. From a technical point of view, a full text index is another auxiliary index that provides an additional entry point to the data, similar to the spatial index. Yet again, it needs to be carefully integrated with a user interface concept. Finally, our architecture does not yet address synchronization with server-based data to achieve better integration with "the cloud".

The remainder of this thesis focuses on the central aspects of the architecture presented in this chapter: Chapter 3 and Chapter 4 solve important challenges in the Data Management Layer of our architecture. Chapter 5 deals with interoperability between devices in an ad-hoc fashion, and Chapter 6 addresses interoperability with web applications.

3

Efficient Attribute Retrieval in RDF Triple Stores

Our architecture builds on a central on-device repository, in which applications share resource and context data, as discussed in Chapter 2. The data is modeled and managed by means of the W3C Resource Description Framework (RDF) and queried by the W3C SPARQL Protocol and RDF Query Language (SPARQL). This enables interoperability across applications at the data management level and even supports complex analytical queries across many different domains. This is a leap forward from the isolated data silos typically found on mobile devices today. However, one must not forget the simple queries, which applications frequently execute. For many tasks, such as displaying all meetings for this week or listing all missed calls, applications only require, in essence, a simple object or record store. I. e. in this case the required functionality is hardly more than selecting a set of resources and retrieving a number of (if not all) attributes that these resources carry. So far, RDF data management research has concentrated on complex analysis of semantic web data, e. g. to discover non-trivial relationships. While simple record-oriented queries were always possible, they were not particularly optimized, as the research focus was elsewhere.

This chapter presents an approach for efficient execution of record-oriented queries in RDF data management systems, as published in [Brodt et al., 2011a]. Our approach exploits storage locality in the existing RDF indexes and offers the query optimizer an alternative query plan that may require significantly less joins. Thus, the optimizer may chose between our approach and the traditional query plans (to which we refer as “canonical plans”), based on the estimated costs of both plans. This way, we do not

at all compromise the power of RDF data management to solve complex analytical queries, but we add the ability to process record-oriented queries efficiently.

First, we give an introduction to RDF, SPARQL and the state of the art concerning RDF data management, i. e., so-called *triple stores* (Section 3.1). Then, we present our approach to retrieve a larger number of attributes for a set of selected resources (Section 3.2). We discuss its integration into the query optimizer (Section 3.3), and introduce a dedicated index structure for attribute retrieval (Section 3.4). We evaluate our concepts (Section 3.5), both on a mobile device and on a desktop computer, before we conclude the chapter (Section 3.6).

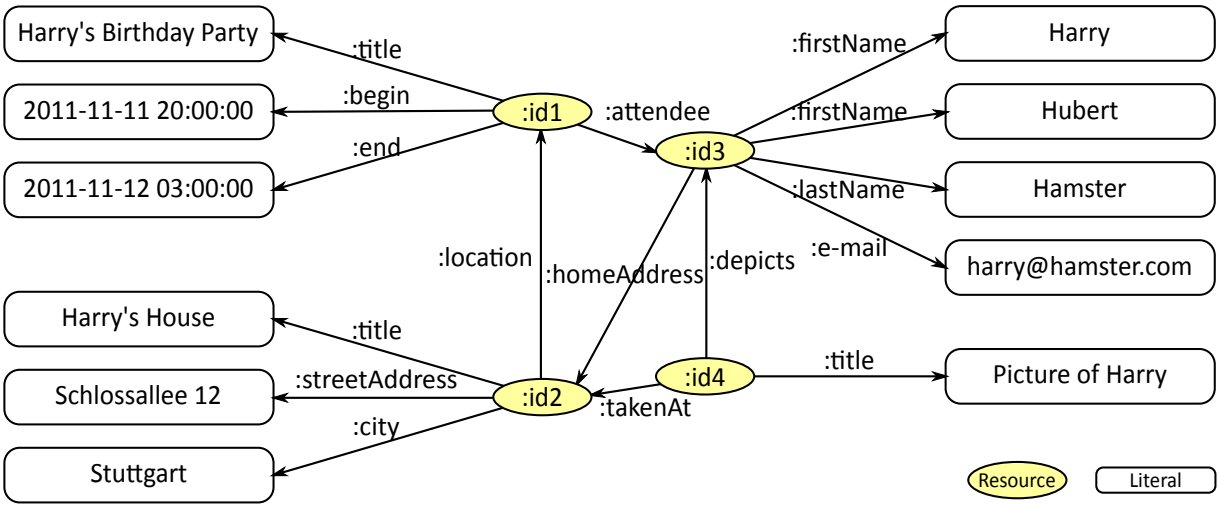
3.1 State of the Art and Foundations

Before we discuss our approach to improve attribute retrieval in queries to RDF data, it is worth taking a closer look at RDF, SPARQL, and the state of the art to implement corresponding data management systems, so-called triple stores. Note that we do not explain all concepts of RDF and SPARQL. Instead, we restrict ourselves to relevant aspects for our approach, without violating general applicability. Later chapters of this document introduce additional concepts as needed.

3.1.1 The W3C Resource Description Framework (RDF)

The W3C Resource Description Framework (RDF) [Klyne and Carroll, 2004] is a data model for structured information that was standardized as a key enabler of the Semantic Web to express metadata on the web. Its flexibility and its strength to model relations between resources lead to a wide adoption in many other application domains including life sciences, Web 2.0 platforms, and data integration tasks. RDF provides great flexibility towards any kind of schema and is even usable without a schema at all. It is very extensible and supports relationships between resources as first-class citizens. RDF allows collecting data in a "pay-as-you-go"-fashion, starting with very little schema information and refining the schema later, as required. Thus, RDF allows breaking the traditional data engineering workflow of designing a schema first and populating it later. This suits RDF for scenarios such as community-based data collection, and in fact large knowledge bases from sources including Wikipedia have been built using RDF [Auer et al., 2007; Suchanek et al., 2008].

Although RDF data is often serialized using XML, the basic data model of RDF consists of *(subject, predicate, object)*-triples: *subject* is a Uniform Resource Identifier (URI) that names the described resource. *Predicate* is a URI that denotes a particular relation or attribute. In case of a relation, *object* is the URI of the resource with which



(a) Graph representation

```
:id1  :title          "Harry's Birthday Party".
:id1  :begin          "2011-11-11 20:00:00".
:id1  :end            "2011-11-12 03:00:00".
:id1  :location       :id2.
:id1  :attendee       :id3.

:id2  :title          "Harry's House".
:id2  :streetAddress  "Schlossallee 12".
:id2  :city           "Stuttgart".

:id3  :firstName      "Harry".
:id3  :firstName      "Hubert".
:id3  :lastName       "Hamster".
:id3  :e-mail         "harry@hamster.com".
:id3  :homeAddress    :id2.

:id4  :title          "Picture of Harry".
:id4  :takenAt        :id2.
:id4  :depicts        :id3.
```

(b) Triples in Turtle notation [Beckett and Berners-Lee, 2011]

Figure 3.1: Sample RDF graph

the subject is related. In case of an attribute, *object* is a literal that denotes a certain value. A literal is usually given as a string and may carry a type that is denoted by yet another URI.

The triples form a directed labeled graph that represents the resources with their relations and attributes. As an example, Figure 3.1(a) depicts an RDF graph of four resources: The event “Harry’s Birthday Party” (id1) taking place at Harry’s House (id2), which Harry (id3) is attending, of course, and a picture of Harry (id4) which was taken at the party. Figure 3.1(b) shows the equivalent triples in Turtle notation [Beckett and Berners-Lee, 2011].

3.1.2 The W3C SPARQL Protocol and RDF Query Language (SPARQL)

SPARQL [Prud’hommeaux and Seaborne, 2008] is a W3C-recommended language to query RDF graphs by graph pattern matching. At its core, a SPARQL query consists of triples containing at least one variable. We refer to such triples as *triple patterns*. The graph pattern expressed by a SPARQL query is made up of conjunctions (and also disjunctions) of triple patterns. The result is a table of variable bindings, such that the pattern exists in the queried RDF graph using the variable bindings in each result row. For instance, the SPARQL query in Listing 3.1 returns the first name, last name, and e-mail address of everybody whose home address is “Harry’s house".

```
SELECT ?first ?last ?email WHERE {
1   ?address  :title        "Harry's House".
2   ?person   :homeAddress  ?address.
3   ?person   :firstName    ?first.
4   ?person   :lastName     ?last.
5   ?person   :e-mail       ?email.
}
```

Listing 3.1: Sample SPARQL query

Executed on the RDF graph of Figure 3.1, the result would be a table with three columns—one column for each variable in the select clause—and two rows, such as shown in Table 3.1, because the semantics of SPARQL compute the cross-product on the two first names of Harry.

3.1.3 RDF Data Management Systems: Triple Stores

Systems to store and query RDF data, so-called *triple stores*, were long of limited interest outside the Semantic Web community and earlier RDF frameworks, such as Jena [Jena], focused on functionality over performance. Only in recent years, database

?first	?last	?email
Harry	Hamster	harry@hamster.com
Hubert	Hamster	harry@hamster.com

Table 3.1: Result of the query in Listing 3.1 executed on the RDF graph in Figure 3.1

research addressed the problem of querying very large RDF datasets efficiently. The challenge of efficient RDF queries originates from the decomposed triple structure and lies in the many join operations that are required to reassemble the data.

A system that manages RDF data cannot assume a defined schema in the general case. Thus, it possesses little knowledge about the higher-level structure of the data, e. g. entity types like “person” or “address”. Hence, it can store, index, and process the data only at triple-level. Also, in a SPARQL query, any part of a triple pattern may be a variable. I. e., triple patterns with variables in different positions require different indexes.

All state-of-the-art approaches [Abadi et al., 2007; Weiss et al., 2008; Neumann and Weikum, 2010; Erling and Mikhailov, 2009; Atre et al., 2010] have in common that they first map all URIs and literals to integer IDs. Internally, the RDF triples are stored, indexed, and queried by these IDs, which is a lot faster than processing entire strings. Only to return a query result, the IDs are mapped back to URIs or literals.

Different indexing techniques for efficient RDF queries have been published. SW-Store [Abadi et al., 2007] creates a two-column table in a column-store database for each RDF predicate and stores the *(subject, object)*-pairs of the RDF triples in the respective tables. Abadi et. al. also propose to materialize the corresponding *(object, subject)*-pairs, but they did not implement this. Virtuoso [Erling and Mikhailov, 2009] uses bitmap indexes for fast bit vector joins. Virtuoso indexes the triples in different permutations and applies different compression schemes on each of them. BitMat [Atre et al., 2010] stores RDF triples in a compressed bit-matrix structure which is accessible for different parts of the triple patterns. It processes the joins by initial pruning followed by a variable-binding-matching algorithm. Hexastore [Weiss et al., 2008] and RDF Triple Express (RDF-3X) [Neumann and Weikum, 2008] take indexing of triples to the extreme: They create a sorted index on all six permutations of the RDF triples: subject-predicate-object (SPO), SOP, PSO, POS, OPS, and PSO. This way, every possible triple pattern can be directly mapped to an index lookup on a triple index. Also, any such index lookup returns a sorted list of triples. These sort orders may be exploited in the joins later on. RDF-3X even indexes aggregations of triples. In case not all parts of a triple are required to process a query, RDF-3X uses one of the six aggregated indexes: SP*, SO*, PS*, PO*, OP*, and PS*. Finally, the so-called

fully aggregated triple indexes project away two columns: S**, P**, and O**. As the (fully) aggregated indexes store the amount of triples in each aggregation, they are also used for cardinality estimation of single triple patterns.

3.1.4 Execution Plans for SPARQL Queries

As stated above, state-of-the-art triple stores keep the triples in a single three-column table with indexes in various (if not all) permutations and aggregations of *subject*, *predicate*, and *object*. In this chapter we assume sorted indexes, such as the B^+-trees used by RDF-3X. Such an index efficiently returns all triples having one or two constant values at a given position,[1] which precisely corresponds to a triple pattern. To combine these triples to the graph pattern defined in the query, joins are required. By carefully choosing indexes on the right triple permutation, index sort orders can frequently be exploited for efficient merge joins. Otherwise, hash joins can be used. We call this approach to generate an execution plan *canonical*, as it can be applied directly on any SPARQL graph pattern.

Figure 3.2 shows a canonical execution plan for the query of Listing 3.1. The query optimizer chose to use the *predicate-subject-object (PSO) index*, for the triple patterns in lines 2, 3, 4, and 5 of Listing 3.1 (other join orderings are of course possible). Thus, the resulting triples are all sorted by their subject, i. e. the person. This allows using three cascaded merge joins. The first triple pattern comes sorted by the address, so a hash join is required to join it with the rest of the query.[2]

Canonical execution plans are relatively easy to build and able to restrict intermediate results early, through a number of measures [Neumann and Weikum, 2009]. This makes them suitable for complex analytical queries on relationships between many resources; for instance, to identify all persons depicted on a picture that was taken at an event taking place in their own home address and who are the CEO of the company that produced the camera with which the picture was taken. However, canonical plans are not necessarily optimal for retrieving attributes of resources. Lines 3, 4, and 5 in Listing 3.1 illustrate that SPARQL requires one triple pattern per attribute to query attributes, such as the first name. Consequently, to retrieve n attributes of a resource, a query must use n triple patterns. A canonical plan evaluates these triple patterns using n index scans and $n - 1$ joins. Even though most often a

[1] Note that triple patterns consisting of three variables are also possible. They can be translated to full scans over a triple index; but without further optimization this is extremely inefficient, as this means scanning the entire database. However, such triple patterns are rare in practice and systems including RDF-3X do not even support them.

[2] Neumann and Weikum [2009] reduce the intermediate results of the cascaded merge joins to tuples that will later find a join partner in the hash join through *sideways information passing*. Thus, the execution plan of Figure 3.2 is not as bad is it may appear having the selective hash join on top.

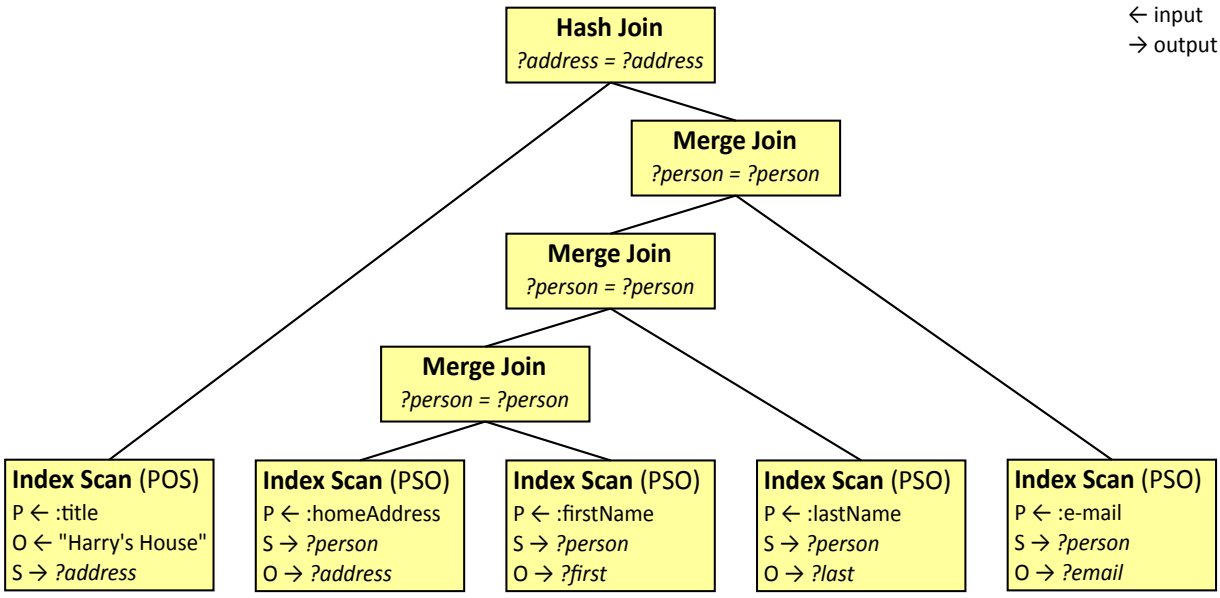


Figure 3.2: Canonical execution plan for the SPARQL query of Listing 3.1, e. g. as generated by [Neumann and Weikum, 2008]

cascade of merge joins will be used to assemble the attributes to larger result tuples, these joins impact query performance severely for larger amounts of attributes.

In a record-based RDBMS, all attributes of a resource are stored contiguously on disk in the same record. Thus, the record can be retrieved in a single fetch operation and does not need to be assembled first. This is not possible for RDF because the schema is unknown. However, the subject-predicate-object (SPO) index is primarily sorted by subject, i. e., it stores all relations and attributes of a resource contiguously. By using the SPO index to retrieve attributes of a resource, a potentially large number of joins can be saved, without compromising the advantages of canonical plans for complex graph patterns.

3.2 Attribute Retrieval Approach

Canonical plans use the predicate-subject-object (PSO) index to retrieve *one* particular attribute, e. g. the first name, for *all* resources in the data set. Contrarily, the subject-predicate-object (SPO) index supports retrieving *all* attributes of *one* given resource without a join on the resource ID for every attribute. This is because the SPO index stores all triples with the same subject (i. e. resource ID) contiguously. On the other hand, it cannot easily identify the relevant triples without knowing the triple subject. The subject is a variable in every triple pattern that retrieves an attribute.

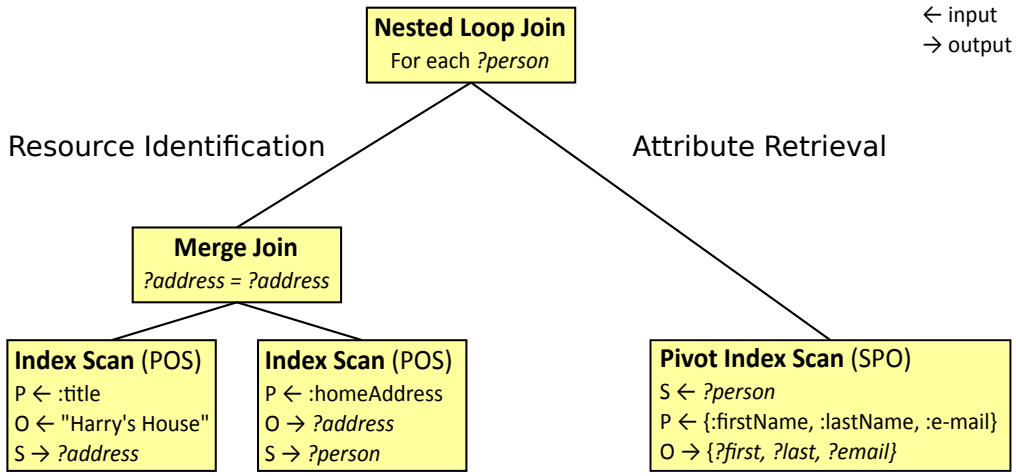


Figure 3.3: Execution plan for the SPARQL query of Listing 3.1 using our approach: It first identifies the resources in question and then retrieves all required attributes using a nested loop join and a pivot index scan

Alternatively, scanning the SPO index top to bottom and joining the result with the remainder of the query is not an option, as this would scan the entire database.

We propose a processing model that splits SPARQL queries into two conceptual parts: *resource identification* and *attribute retrieval*. Resource identification determines the set R of qualifying resources. For each resource $r \in R$, attribute retrieval fetches the values of the attribute set A, as mandated by the query. E. g. for the query of Listing 3.1, R consists of all persons who live in "Harry's House" and A is {*firstName, lastName, e-mail*}.

3.2.1 The Pivot Index Scan Operator

To implement attribute retrieval using the SPO index, we designed the *pivot index scan* operator. It retrieves the values for a given resource r and attribute set A from the SPO index and pivots them into one or more result tuples in a single operation. The pivot index scan is designed for use under a nested loop join which connects resource identification with attribute retrieval. The nested loop join iterates on all identified resources, invokes the pivot index scan on each resource, and propagates all its result tuples. For the query of Listing 3.1, this results in an execution plan as shown in Figure 3.3: Resource identification determines the set of qualifying resources R as the identifiers of all persons living in "Harry's House". A nested loop join invokes attribute retrieval on each person: a pivot index scan to look up the attribute set A from the SPO index, i. e. the first name, the last name, and the e-mail address of each person.

```
▷ B-tree search for resource r and smallest attribute a_min ∈ A
scan ← SPO_Index.openScan(r, a_min)

▷ Retrieve and buffer all required attribute values
(s, p, o) ← scan.firstTriple()
for all a ∈ A do                                   ▷ A is sorted in index order
    while s = r ∧ p < a do
        (s, p, o) ← scan.nextTriple()              ▷ Skip intermediate triple
    end while

    if (s > r ∨ p > a) ∧ ¬isOptional(a) then                              ▷ (*)
        exit                          ▷ r does not have all required attributes
    end if

    ▷ Buffer all values for a
    repeat
        valueBuffer[a] ← valueBuffer[a] ∪ {o}                            ▷ (**)
        (s, p, o) ← scan.nextTriple()
    until s > r ∨ p > a
end for
```

Listing 3.2: Implementation of the pivot operation, which drives a scan on the SPO index (for a simpler presentation we assume that the scan always returns a triple)

As shown in Figure 3.3, the pivot index scan takes a variable binding as input, which was produced by resource identification. During one run of the nested loop join, this variable binding is treated as a constant: it is the subject of the triples to look up from the SPO index. The second input for the pivot index scan is the set of attributes A, as defined in the query. Thus, the pivot index scan fetches a set of attributes for a given resource in a single invocation. Figure 3.3 shows that it possesses a set of output variables for this purpose: one for every attribute to retrieve.

Unlike other operators, the pivot index scan processes *sets* of triples rather than individual triples. Internally, it consists of two parts; an index scan and a pivot operation. First, the index scan looks up the first triple containing the resource r as subject, and the smallest attribute $a_{min} \in A$ as predicate from the SPO index. This requires the attributes in A to be sorted in the same order as they appear in the index. Subsequently, the pivot operation makes the index scan iterate through the triples of r, until the values for all attributes in A have been fetched, or until all triples about r have been read. Listing 3.2 illustrates how the pivot operation selects all triples of

the resource containing a predicate $p \in A$. The pivot operation projects the objects of these triples and buffers them.

3.2.2 Optional Attributes

SPARQL allows declaring parts of the query *optional*. This corresponds to a one-sided outer join in the relational model. I. e., the optional parts of a query will be evaluated and returned if they are found. Otherwise the projected variables from an optional query part produce NULL values, but do not cause a result tuple to be dismissed. If single triple patterns which retrieve attributes are declared optional, they can be evaluated in a pivot index scan. The pivot index scan operator would normally abort if it cannot find a triple containing a queried attribute for the current resource. To handle optional attributes, the query optimizer only needs to tell the pivot index scan which attributes are mandatory and which are optional. Listing 3.2 shows how a pivot index scan handles missing optional or mandatory attributes in the line marked with an asterisk (*).

3.2.3 Multi-Attributes

It is not at all safe to assume that a resource carries only one value for a particular attribute. There can be an arbitrary number of triples with the same subject and predicate, but with a different object. In this case, the predicate is a multi-attribute and the semantics of SPARQL require producing the cross product. For instance, the subject `id4` in Figure 3.1 carries two triples with predicate `firstName` and one triple with predicate `lastName`. Consequently, the query of Listing 3.1 produces two result tuples; one for each first name. It is important to note that the result of a pivot index scan is not one tuple, but a set of tuples.

A canonical plan automatically produces the cross product of multi-attributes when the joins assemble the attributes to larger result tuples. To preserve this behavior, the pivot index scan must buffer all values for each attribute $a \in A$ when it scans the triples of one resource. It must then calculate the cross product over these buffers in an explicit separate step. Listing 3.2 shows how a pivot index scan buffers the values indexed by attribute in the line marked with a double-asterisk (**). In a Volcano-style [Graefe, 1994] operator implementation providing an open/next/close interface, the `open()` call has to scan the relevant triples and buffer the attribute values, as shown in Listing 3.2. It may then close the underlying index scan. `next()` then returns the next combination of the cross product. The `close()` call discards the buffer.

```
?person :wasOrIsMarriedWith ?spouse1.
?person :wasOrIsMarriedWith ?spouse2.
```

Listing 3.3: Triple patterns extending Listing 3.1 by a cross product on all values of one attribute

3.2.4 Multiply Selected Attributes

SPARQL allows selecting an attribute of a resource more than once. This is particularly interesting in combination with multi-attributes and produces the cross product of all attribute values. For example, Listing 3.1 could be extended by the triple patterns in Listing 3.3 to return the cross product on all (ex-) wives of the person.

This might seem a purely academic case at first, but in fact occurs not infrequently. The cross product on multiply selected attributes is often used to find a particular relation between all values. It is usually combined with a filter to ensure that resulting value pairs contain different values, i. e., `?spouse1 != ?spouse2` in the example. This way, one could determine whether two (ex-) wives of the person share the same hobby or hair-color, for instance. Naturally, it is perfectly possible to simply query the cross product of the values as (part of) the query result.

A canonical plan will strictly map every triple pattern of the query to an index scan and create joins between them. It will equally apply this principle to triple patterns which retrieve the same attribute and the joins will automatically produce the cross product on the respective values.

A pivot index scan must replicate this behavior explicitly. As discussed above, it must buffer the values of multi-attributes and produce their cross product anyway. To create the cross product on a multiply selected attribute a, the query optimizer must inform the pivot index scan about the number k_a denoting how often to consider the values for a in the cross product. The pivot operation can be used as shown in Listing 3.2. Only the cross product algorithm must consider the value buffer for a not just once, but k_a times.

Formally, this means that the set A of attributes to retrieve is really a *multi-set* in the general case. In the interest of a simpler presentation of this paper, we will nevertheless refer to A as a set. Consequently, by $|A|$ we do not really mean the number of distinct attributes to retrieve, but the number of triple patterns involved in attribute retrieval. This does not restrict the generality and applicability of our approach.

3.2.5 Related Work

The idea of a pivot operation transposing rows into columns is not new. Some RDBMS including Microsoft SQL server provide the *Pivot* and *Unpivot* data manipulation operators. Pivot transforms a set of rows into a set of fewer rows with additional columns. Unpivot is the reverse operation of Pivot [Cunningham et al., 2004]. Wyss and Robertson [2005] provide a formal characterization. When the set of columns is unknown in advance, a table can be implemented as a *property table* that is accessed using the Pivot operator. This is more flexible, as columns can be added as property rows of the form (id, propertyname, propertyvalue). This form is very similar to the RDF data model. Before Pivot and Unpivot were introduced, application programmers frequently used complex SQL statements with a nested subquery for every pivoted column [Cunningham et al., 2004]. These queries were inefficient and hard to optimize, and strongly resembled canonical execution plans for SPARQL queries.

The main difference to our pivot index scan operator is that Pivot must not produce multi-attributes, as the relational model does not allow them. For this purpose, Pivot takes an aggregation function which collapses multi-attributes into a single value. SPARQL, by contrast, requires returning the cross product on all multi-attribute values.

3.3 Plan Generation

The classical bottom-up dynamic programming approach of System R [Selinger et al., 1979] is well-suited to create canonical execution plans for SPARQL graph patterns. We first explain bottom-up plan generation for canonical plans, as implemented in [Neumann and Weikum, 2008]. Then we introduce our approach to extend such an optimizer so that it produces execution plans that use pivot index scans, if this is cheaper. We describe our cost model and an approach to cardinality estimation in presence of multi-attributes. Finally, we address selective attributes, which influence the set of selected resources in the query result.

3.3.1 Generating Canonical Plans

As illustrated in Figure 3.2, a graph pattern containing n triple patterns can be solved by a canonical plan which consists of n index scans and $n-1$ joins. Such a plan can be created using the following recursive rule: A plan which covers one triple pattern is an index scan. A plan covering $x > 1$ triple patterns is a join of two sub-plans covering a and b distinct triple patterns, respectively, with $a + b = x$. A plan covering $x = n$ triple patterns solves the query.

The canonical optimizer uses a table for dynamic programming, which contains all sub-plans produced so far. Thus, the table consists of n rows, where all plans in row x cover exactly x triple patterns. The plans of each row are grouped in sets of plans solving the same problem, i. e. covering the same triple patterns. First, the optimizer seeds the first row of the dynamic programming table with n sets of index scans. For every triple pattern, it generates an index scan on the index of every possible triple permutation, such that all possible sort orders are considered. E. g. for the second triple pattern of Listing 3.1, an index scan on the predicate-subject-object (PSO) index produces the triples sorted by person, whereas using the POS index orders them by address. Subsequently, the higher rows are populated incrementally. Every plan in row x is produced by joining one sub-plan from row a and one sub-plan from row b, with $a + b = x$. The two sub-plans must share a common variable to join on and their problems must not overlap, i. e., their sets of covered triple patterns must be disjoint. If both sub-plans are sorted on the join variable, a merge join is used; otherwise a hash join is created. As explained in [Neumann and Weikum, 2008], the optimizer should only accept a new plan for a problem, if it is cheaper than all other plans of the same problem, or if it introduces a new sort order. Finally, the optimizer chooses the cheapest plan in row n.

3.3.2 Generating Plans with Pivot Index Scans

Before a pivot index scan can be created, the triple patterns which retrieve attributes have to be identified. In order to be evaluated in a pivot index scan, the triple patterns must comply with the following requirements:

1. The triple patterns must share the same subject.
2. The subject must be a variable.
3. The predicate must be a constant.
4. The object must be a variable.
5. The object must be listed in the `SELECT` clause of the query.
6. The object must not occur in any other triple pattern.
7. There must be at least two such triple patterns.

The first four requirements state that the triple patterns retrieve the values for a list of fixed attributes from the same resources. Requirement 5 checks that every value is

really contained in the query result—if this was not the case, the value might not be needed at all. Requirement 6 ensures that no triple pattern is involved in a join on the object. Joins on the object are likely to restrict the query result and thus strongly indicate that a triple pattern belongs to the resource identification part of the graph pattern. Finally, a pivot index scan would not save any joins and not benefit from storage locality at all, if it retrieved only a single attribute (requirement 7).

To identify qualifying triple patterns, we sort the triple patterns by subject, predicate, and object. We order variables in the subject before constants, constants in the predicate before variables, and variables in the object before constants. This causes a grouping of triple patterns that may be evaluated in a single pivot index scan. As a side product, this sort order also ensures that the set A of attributes to retrieve is sorted in the same order as the attributes appear in the SPO index. This is later required by the pivot index scan, as described in Section 3.2.

Every qualifying set of triple patterns for attribute retrieval can be solved by one pivot index scan. For each set, the predicates of the triple patterns constitute one set A and the pivot index scan covers $|A|$ triple patterns. As explained above, a canonical optimizer joins existing sub-plans to larger plans, starting with index scans as atomic plans. A pivot index scan is an atomic plan as well, but it covers more than one triple pattern. Thus, in addition to seeding the first row of the dynamic programming table with index scans, the optimizer must also generate pivot index scan s in row $|A|$ for each set of triple patterns. When the bottom-up algorithm reaches row $x = |A| + b$, it generates a join between the pivot index scan and all sub-plans in row b. Naturally, these sub-plans must contain the variable which denotes the resource for which the pivot index scan retrieves the attributes. In addition to that, the optimizer must choose a nested loop join instead of a merge or hash join when it joins a pivot index scan. Also, the pivot index scan must become the right child of the join, i. e., it must occur inside the loop.

The nested loop join should occur near the root of the plan. It does, ideally, not restrict the cardinality of the query result (see Section 3.3.5), but might increase it through cross products quite a bit. Thus, executing the pivot index scan too early in the query plan causes an unnecessarily high number of intermediate result tuples. Moreover, it is in practice not invalid to assume the joins of the resource identification part of the query to be rather selective. Thus, query plans with early pivot index scan s will be rated more expensive and eliminated by plans which use it later. The optimizer could, however, anticipate this and include the pivot index scan s only towards the end. The bottom-up algorithm could ignore them until it reaches row $x = n - \sum_i \left|A_i\right|$, where $\sum_i \left|A_i\right|$ denotes the maximal amount of triple patterns covered by all possible pivot index scan s.

3.3.3 Cost Model

A pivot index scan executes one lookup in the SPO index for every resource r that is produced by resource identification. Even if r does not carry all mandatory attributes, the pivot index scan only removes r from the query result *after* the index lookup has been performed, as illustrated in Listing 3.2. We do not generally assume that the amount of scanned triples for each resource is significant. A pivot index scan rarely needs to access more than one leaf page of the SPO index for a single resource, as the triples can be compressed very well (we experienced about 4 000 compressed triples in a single leave page using the method presented in [Neumann and Weikum, 2008]). The index lookup costs for each resource will by far dominate the costs for sequentially scanning the respective triples. In addition, the cost of the nested loop join operator itself is negligible. Thus, if resource identification is estimated to produce $|R|$ resources with total costs $cost_{res.id.}$ and one index lookup causes costs of $cost_{lookup}$, then the costs of the nested loop join over resource identification and a pivot index scan $cost_{nl}$ can be simply estimated as follows:

$$cost_{nl} = cost_{res.id.} + |R| * cost_{lookup}$$

This shows that plans using a pivot index scan are practically independent of the number of selected attributes $|A|$, but in turns do depend on $|R|$. Thus, it depends on the query and on the RDF data set, whether a pivot index scan should be favored over a canonical plan. Our plan generation approach always creates both possibilities and lets the cost model decide which plan is the cheapest.

3.3.4 Cardinality Estimation

Estimating the cardinality of a plan that uses our attribute retrieval approach is no different from general cardinality estimation, as cardinality depends on the queried data and not on the chosen execution plan. A pivot index scan evaluates a star-shaped graph pattern which comprises many triple patterns. Thus, a method to estimate cardinality of larger subgraphs is desirable, rather than estimating single triple patterns and combining them incrementally.

Characteristic sets

Characteristic sets [Neumann and Moerkotte, 2011] provide a good approximation for star-shaped subgraphs in RDF data. Characteristic sets describe resources through their predicates, to characterize them in a similar way as entity types would. Thus,

characteristic sets can be seen as some sort of "soft entity types". We introduce them here, as we will refer to them in the next section, too.

A characteristic set $S_C(s)$ of a triple subject s occurring in a set of RDF triples $\mathcal{T}$ is defined as the set of predicates which are connected to s:

$$S_C(s) = \{p \mid \exists\, o : (s, p, o) \in \mathcal{T}\}$$

The set of all characteristic sets occurring in $\mathcal{T}$ is consequently defined as follows:

$$S_C(\mathcal{T}) = \{S_C(s) \mid \exists\, p, o : (s, p, o) \in \mathcal{T}\}$$

Neumann and Moerkotte observed that the amount of characteristic sets in a real-world RDF data set is surprisingly low. They compute and store all characteristic sets a set of RDF triples $\mathcal{T}$ and count the resources belonging to each characteristic set S_C, denoted by $count_{S_C}(Res)$. To estimate the number of resources matching a star-shaped graph pattern, all characteristic sets have to be determined that fully include the predicates of the pattern. Summing up $count_{S_C}(Res)$ for all these characteristic sets, returns the number of resources matching the pattern.

The output cardinality equals this number of resources, conditioned there are no multi-attributes. Otherwise, the cross product over multi-attributes must be taken into account. To estimate multi-attributes, Neumann and Moerkotte compute $count_{S_C}(p)$ for each predicate p as the total sum of how often p occurred in a resource belonging to S_C. Thus, the average occurrence of p in a resource belonging to S_C is

$$\frac{count_{S_C}(p)}{count_{S_C}(Res)}.$$

For instance, in the RDF graph in Figure 3.4, three resources (id3, id5, and id6) belong to a characteristic set S_C that fully includes $\{firstName,\ lastName\}$. I. e., $\sum_{S_C} count_{S_C} = 3$. Within these three resources, the *firstName* predicate occurs four times: two times for Harry, once for Bernie, and once for Hermine. Thus, $\sum_{S_C} count_{S_C}(firstName) = 4$. This means that, on average, these resources carry $\frac{4}{3}$ first names.

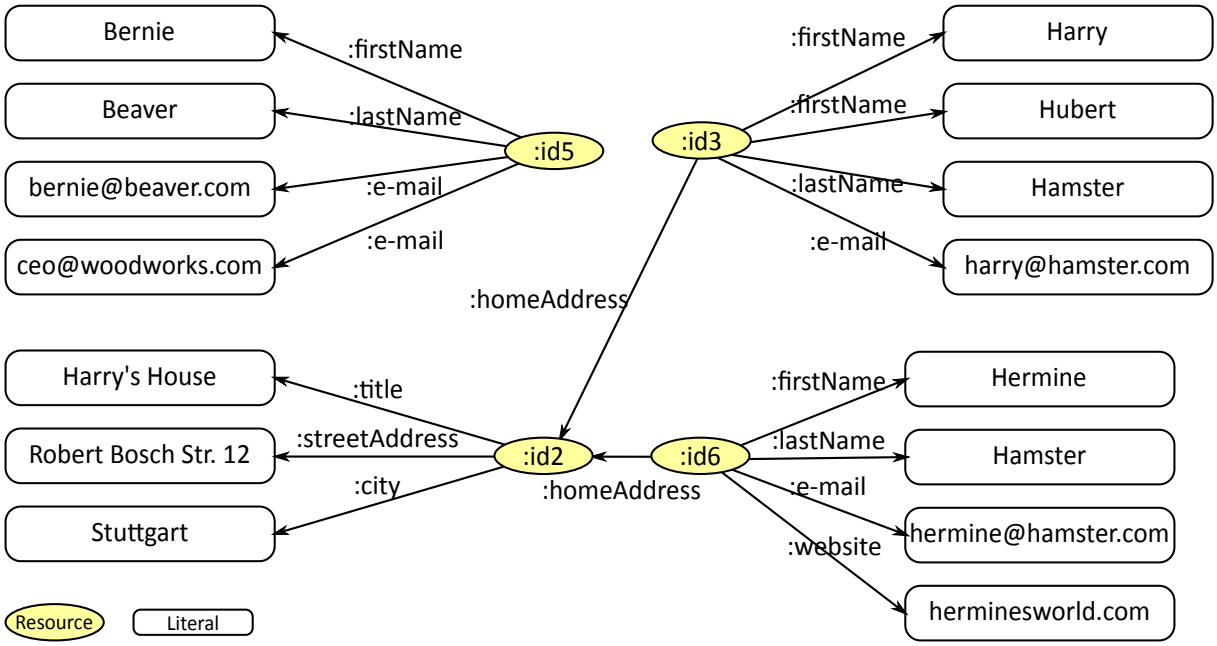


Figure 3.4: Sample RDF graph

Estimating Output Cardinality of Pivot Index Scans using Characteristic Sets

To estimate the output cardinality of a pivot index scan with subject variable ?s, the set $P_Q(?s)$ of predicates which are connected to ?s in the query Q has to be determined:

$$P_Q(?s) = \{p \mid \exists\, o : (?s, p, o) \in Q\}$$

Is is important to note, that $P_Q(?s)$ must also include predicates which are not retrieved by the pivot index scan, but still connected to its subject variable in the query. Resources which do not possess these predicates will never reach the pivot index scan and are therefore irrelevant for output cardinality. Thus, for the query in Listing 3.1, $P_Q(?person)$ is {*firstName, lastName, e-mail, homeAddress*}, even if the pivot index scan does not retrieve the attribute *homeAddress*. But every resource that is contained in the final query result must have a home address, nevertheless. In the data set of Figure 3.4, *id5* also possess all attributes retrieved by the pivot index scan. Considering *id5* for cardinality estimation would lead to an overestimation of resource identification. Also, the average multi-attribute cardinality of the *e-mail* attribute would be overestimated, due to the second e-mail address of *id5*. However, *id5* is not considered in cardinality estimation, as it misses the *homeAddress* attribute.

Resource identification will not return *id5*, so that it can impossibly be contained in the query result.

After $P_Q(?s)$ has been computed, all characteristic sets are determined which fully include $P_Q(?s)$. In Figure 3.4 these are {*firstName, lastName, e-mail, homeAddress*} and {*firstName, lastName, e-mail, homeAddress, website*}. The occurrence counts of every predicate $p \in P_Q(?s)$ are summed up to $count(p)$ across all these characteristic sets, and their resource counts to $count(Res)$:

$$count(p) = \sum_{S_C \supseteq P_Q(?s)} count_{S_C}(p)$$

$$count(Res) = \sum_{S_C \supseteq P_Q(?s)} count_{S_C}(Res)$$

We estimate the output cardinality of the pivot index scan as $count(Res)$ times the factor by which the cross-product over multi-attributes increases the output. This factor can be estimated by multiplying the average occurrence of each predicate $p \in P_Q(?s)$, as described above. This, however, does not take multiply selected attributes into account, as discussed in Section 3.2.4. Thus—and this is a contribution beyond the cardinality estimation approach of Neumann and Moerkotte [2011]—every predicate must be considered in the cross-product estimation as often as the query selects it. I. e., in the product, the average occurrence of each predicate in the characteristic sets considered must be multiplied by its occurrence k_p in the query. Finally, we estimate the cardinality of a star-shaped graph pattern with subject variable $?s$ as

$$card(?s) = count(Res) * \prod_{p \in P_Q(?s)} \frac{k_p * count(p)}{count(Res)}.$$

In the example of the query of Listing 3.1 executed on the data set of Figure 3.4, the number of selected resources is $count(Res) = 2$. No attributes are multiply selected, so k_p is alyways 1. There are totally three occurrences of the *firstName* attribute, all other attributes occur twice. Thus, the cardinality of the pivot index scan is estimated to be 3, which is correct.

$$card(?person) = 2 * \frac{1*3}{2} * \frac{1*2}{2} * \frac{1*2}{2} = 3.$$

3.3.5 Selective Attributes

The basic assumption of our approach is that resource identification, which determines the set R of qualifying resources, is highly selective. At the same time, attribute retrieval simply fetches the queried attribute values for each resource $r \in R$, but is assumed to influence R very little. While this assumption intuitively holds for many cases, it is far from a given in the general case. It is not true, if only small fraction of R carries certain attributes of A, i. e. the *existence* of attributes is selective.

If, e. g., the triple pattern of Listing 3.4 was added to the SPARQL query of Listing 3.1, it would not return any results for the data set of Figure 3.1 or 3.4. Whereas most people possess at least one first name, a last name, and an e-mail address, most people do not own a helicopter of *any* model at all. Thus, `ownsHelicopterModel` is a very selective attribute.

In a canonical execution plan, the joins between the index scans that retrieve the attributes will eliminate all resources that lack a mandatory attribute. The optimizer will arrange joins over infrequent attributes early in the plan. And through sideways information passing [Neumann and Weikum, 2009], operators higher in the query plan may skip non-qualifying resources early. By contrast, a pivot index scan will only realize that a resource r lacks a mandatory attribute, when it has already found r in the SPO index—and the index lookup is the expensive part. Unlike the cascaded joins of a canonical plan, which are able to influence each other in a zig-zag fashion, a pivot index scan has no chance to influence resource identification.

Execution Plans for Selective Attributes

The only way to avoid failing index lookups in a pivot index scan due to selective attributes is to invoke it only on resources that possess all mandatory attributes in the first place. This means that resource identification must restrict R to such resources. The obvious solution is to remove a selective attribute a_{sel} from the set A of attributes which the pivot index scan retrieves, as shown in Figure 3.5. Instead, a_{sel} is processed in the manner of a canonical plan: using a separate index scan on the PSO index and one join. The join is able to eliminate resources that lack a_{sel} early, so that they never reach the pivot index scan. We refer to this approach as “reduced” in our evaluation.

The separate index scan will return only resources which possess a_{sel}. It will also retrieve the corresponding attribute values. On the other hand, if the optimizer still

```
?person :ownsHelicopterModel ?model.
```

Listing 3.4: Example of a (presumably) selective triple pattern

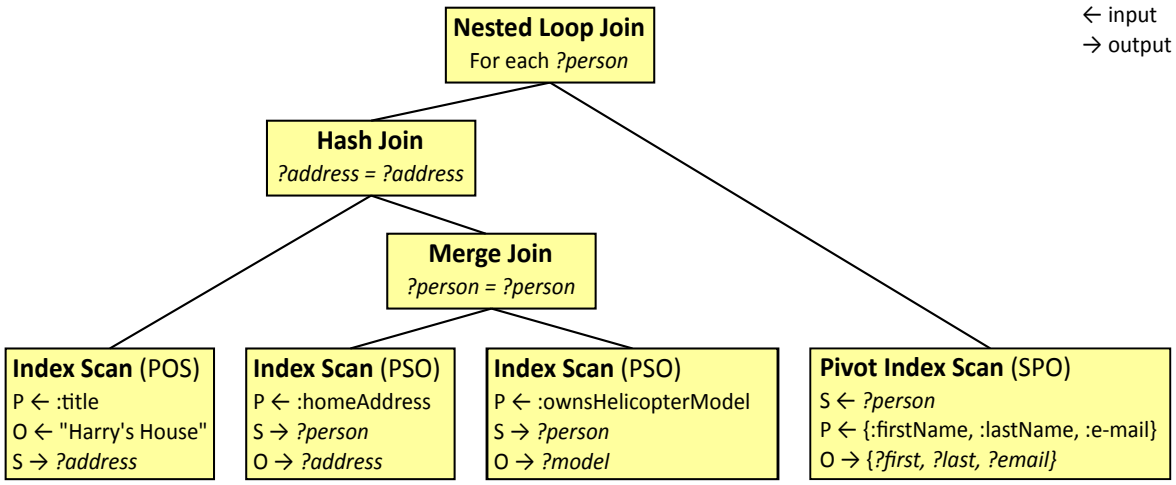


Figure 3.5: "Reduced" execution plan retrieving a selective attribute (ownsHelicopter-Model) in the manner of a canonical plan

chooses to use a pivot index scan, the latter will also come across the attribute values for a_{sel}, practically for free. It loads the memory area that contiguously stores all attributes of a selected resource, so that it is very likely, that it loads the value for a_{sel} as well. So instead of removing the entire triple pattern that includes a_{sel} from the pivot index scan, it would be enough to ensure that every resource that enters attribute retrieval really possesses a_{sel}—without retrieving the value. For this, an index that only includes the predicate and the subject, but lacks the object value, would fully suffice.

As explained in Section 3.1.3, the RDF-3X triple store [Neumann and Weikum, 2008] creates, in addition to the six possible triple permutations, also six *aggregated* indexes (SP*, SO*, PS*,PO*, OS*, and OP*). The aggregations project away one column of the triples and store the amount of resulting duplicates. For instance, the PS* index stores the number of triples in the data set with a particular predicate and subject. RDF-3X uses the aggregated indexes to evaluate triple patterns containing variables that are not used anywhere else in the query, i. e. their values are not required to determine the query result. RDF-3X also uses the aggregated indexes for cardinality estimation of single triple patterns. Naturally, they are smaller than the full triple indexes.

Thus, instead of removing a_{sel} from the pivot index scan completely, as the "reduced" approach would, we can also add an index scan on the aggregated PS* index for a_{sel} to resource identification, as shown in Figure 3.6. The aggregated index scan returns less intermediate results than the full triple index and thus produces less join costs. Nevertheless, it is still able to restrict R to resources which possess a_{sel}. Finally,

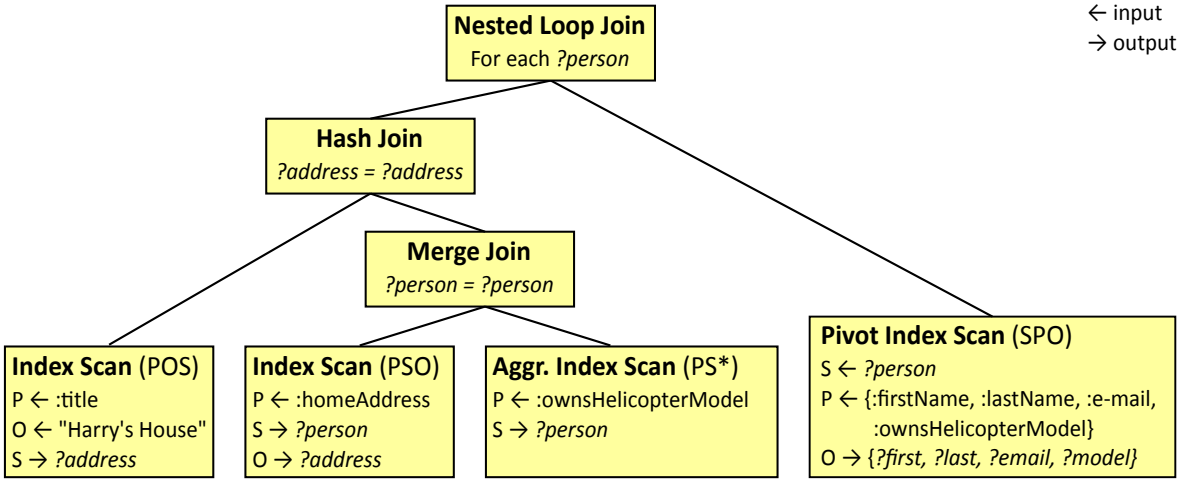


Figure 3.6: "Checked" execution plan using a pivot index scan to retrieve a selective attribute (ownsHelicopterModel): An aggregated index scan selects only resources which do carry the selective attribute

the pivot index scan will retrieve the values for a_{sel}, together with all other attribute values. We call this approach "checked", as the aggregated index scan checks the existence of the selective attribute.

Identifying Selective Attributes

To identify selective attributes, we need to characterize the resources that *enter* attribute retrieval, i. e. the bindings which resource identification creates for the subject variable of the pivot index scan. For this we take a look at the set P of predicates that are connected to this subject variable in the query, but not contained in the attribute list of the pivot index scan; i. e. $P \cap A = \emptyset$. For the query in Listing 3.1, P would be $\{homeAddress\}$. In order to qualify for resource identification, a resource r must possess all predicates $p \in P$. Put differently, if r lacks any of these predicates, it never reaches attribute retrieval. Next, we need a method to determine, which other predicates these resources typically carry, once they possess all predicates $p \in P$. I. e. we need to characterize the resources based on the predicates in P. This is exactly what characteristic sets [Neumann and Moerkotte, 2011] do, as explained in Section 3.3.4.

First, we need to retrieve all characteristic sets in the RDF data set $\mathcal{T}$ that fully include P. By summing up the number of resources belonging to these sets, $count_{S_C}(Res)$, we obtain an upper bound $card_P$ for the input cardinality $|R|$ of the pivot index scan.

$$card_P = \sum_{\substack{S_C \supseteq P \\ \wedge S_C \in S_C(\mathcal{T})}} count_{S_C}(Res)$$

To see how each attribute $a \in A$ influences cardinality, we can similarly calculate the amount of resources which possess a, as well as all predicates in P:

$$card_a = \sum_{\substack{S_C \supseteq \{a\} \cup P \\ \wedge S_C \in S_C(\mathcal{T})}} count_{S_C}(Res)$$

If $card_a \ll card_P$, then a is selective with respect to resources possessing all predicates in P. This means, that a pivot index scan likely performs many index lookups in vain, because many resources entering attribute retrieval lack a.

This approach only works if the query contains other triple patterns with constant predicates for the subject variable of the pivot index scan (otherwise $P = \emptyset$). If this is not the case, it is difficult to characterize the resources that enter attribute retrieval. Assuming statistical independence, we can still roughly determine selective attributes based on their general occurrence in the entire data set. For this we need to determine how many resources carry a particular attribute.

We can obtain this information using characteristic sets as well. For every retrieved attribute $a \in A$, we only need to determine every characteristic set S_C that includes a. By summing up $count_{S_C}(Res)$ for all these sets, we obtain the number of resources in the data set that possess the attribute a. If one attribute returns significantly fewer resources than others, it is selective with respect to the entire data set.

The second approach assumes that resource identification selects resources that behave similarly to all other resources in the data set. In practice, selectivity of an attribute may depend quite heavily on the particular resources in question. E. g. the attribute `ownsHelicopterModel` is utterly selective for people in general, but much less for helicopter pilots or millionaires. Thus, we try to identify selective attributes using the characteristic sets containing the predicates that are not retrieved by a pivot index scan. If this is not possible, we use general occurrence of each retrieved attribute in the data set.

3.4 Attribute Retrieval Index

Most RDF triple stores map the actual values of the triples to integer IDs, as they can be stored and processed much more efficiently than the strings, of which these values mostly consist [Abadi et al., 2007; Neumann and Weikum, 2008; Erling and Mikhailov, 2009]. As a consequence, a dictionary must map every query result back to its string representation before it can be returned. This mapping step may cause significant overhead for large query results; for some queries we observed that over 90% of the total query execution time was spent by the dictionary.

Our attribute retrieval approach exploits locality of RDF triples describing the same resource in the SPO index. We retrieve the attribute values from there to produce the final query result. Due to condition (6) in Section 3.3.2, these values cannot be involved in further operators of the query, i. e. they are simply returned for output. To save the costs of mapping the results back to their external string representations, we implemented a variant of the SPO index that does not store the integer IDs for the object values. Instead, it directly points to the external representation of the object value. Subject and predicate are still represented as IDs, because they are required for query processing and because they are not an output of the pivot index scan. As a pivot index scan never searches by object value, the inner B^+-tree nodes of this attribute retrieval index are organized exactly as in the aggregated SP* index. I. e., they omit the triple object. Thus increases their fan-out by 25%.

The leaf nodes are specifically designed for the access pattern of a pivot index scan. The access pattern starts with the subject representing the resource in question and looks for its list of predicate/object pairs. After the predicate/object pairs have been found, the subject is no longer required. Thus, as shown in Figure 3.7 we store all subjects contiguously at the beginning of a leaf node. After each subject we store the length (in bytes) of the subject's predicate/object list. We store the predicate/object list of each subject contiguously at a location further behind in the page. All predicates and objects are stored in the same page for all subjects, except for the last one. The last subject continues on the next page, if the page header indicates this. The page header also contains the in-page byte offset that points to the beginning of the first predicate/object list. At the same time, this offset marks the exclusive end of the subject list.

We use a simple compression technique for the leaf nodes. We use delta encoding for the subject list and for the predicates of each subject. Both are sorted integer IDs, which allows storing only the difference to their predecessor. This results in smaller integer numbers, which we store using 7-bit variable-length encoding. In most cases, this requires only one byte per subject or predicate. To point to the object values, we

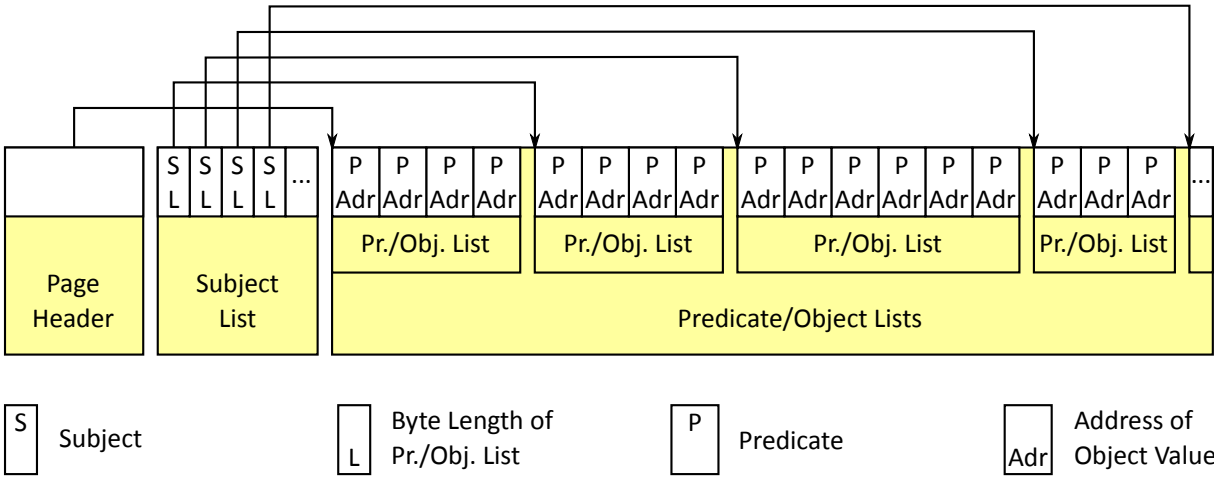


Figure 3.7: Leaf page layout of the attribute retrieval index

store the uncompressed page number and the in-page offset that identify the location where the dictionary stores the external value representation.

The storage addresses of the object values are longer than the internal IDs of the objects. Moreover, the addresses are not sorted. This makes the triples harder to compress than in the fully sorted SPO index. In our experiments, the compression rate was worse by a factor of about 2.2, compared to the compression method for the SPO index presented in [Neumann and Weikum, 2008]. Nevertheless, our compression technique for the attribute retrieval index stored the triples in roughly 50% of their uncompressed size. In addition, the attribute retrieval index is only intended for point queries, so the absolute index size is not primarily an issue. For point queries, it is much more important to access the relevant parts of the leaf nodes quickly, at which the attribute retrieval index excels.

When a pivot index scan searches for the attributes of a resource r, it first locates the right leaf node through standard B-tree search, as illustrated in Listing 3.5. It opens the leaf node and reads the offset pointing to the end of the subject list pos_{slEnd} from the page header. Then, it reads the (sorted) subject list sequentially until it finds the subject. The compressed subjects do not occupy much space, so that the linear search benefits from good processor cache locality and can exploit hardware branch prediction well. For each subject, the pivot index scan reads two variable-length encoded numbers: the difference δ to the preceding subject and the length l of the corresponding predicate/object list. Both numbers are summed up. The sum of the deltas results in the current subject s. The end of the subject list pos_{slEnd} plus the sum of the lengths returns the in-page offset where the predicate/object list of the current

▷ B-tree search for the leaf page by resource r and smallest attribute $a_{min} \in A$
$page \leftarrow AttributeRetrievalIndex.findLeafPage(r, a_{min})$

▷ Determine the position of the subject list: behind the page header
$pos_{reader} \leftarrow PAGE\_HEADER\_SIZE$
▷ The subject list ends where the first predicate/object list starts.
  ▷ This position is stored at a known offset o_{slEnd} in the page header
$pos_{prObjList} \leftarrow pos_{slEnd} \leftarrow read(page, o_{slEnd})$

▷ Read the subject list up to r :
  ▷ s is the current subject
  ▷ l is the length of the corresponding predicate/object list
$(s, l) \leftarrow read(page, pos_{reader})$
while $s < r \wedge pos_{reader} < pos_{slEnd}$ **do**
  $pos_{prObjList} \leftarrow pos_{prObjList} + l$
  $(\delta, l) \leftarrow read(page, pos_{reader})$
  $s \leftarrow s + \delta$
end while

▷ The predicate/object list of r is located between $pos_{prObjList}$ and $pos_{prObjList} + l$

Listing 3.5: Finding the predicate/object list of a given resource r in the leaf page of the Attribute Retrieval Index

subject starts. It ends l bytes further behind. After the pivot index scan has found the predicate/object list, it processes it as shown in Listing 3.2.

3.5 Evaluation

We carried out an extensive performance evaluation both on a desktop computer and on a mobile device to compare (1) canonical plans with (2) execution plans that utilize our attribute retrieval approach on the "ordinary" SPO index and (3) on the attribute retrieval index.

3.5.1 Implementation

We implemented our attribute retrieval approach and integrated it into the RDF-3X [Neumann and Weikum, 2008] triple store, version 0.3.4, which is available as open source.[3] However, our approach is applicable for any triple store that indexes RDF

[3] `http://www.mpi-inf.mpg.de/~neumann/RDF3x/`

data on a per-triple basis in several permutations (including the subject-predicate-object permutation, of course), which is the current state of the art.

In addition to adding a new logical and physical database operator, we had to modify the query optimizer, as described in Section 3.3.2. To integrate the attribute retrieval index, we also had to modify data import, and—most notably—large parts of the runtime system and the dictionary. As a consequence of the attribute retrieval index, intermediate query results no longer consist of dictionary IDs only. Instead, they may consist of IDs and addresses of object values. The runtime system must differ between both types, which makes the code quite a bit more complicated.

3.5.2 Test Setup

We generated artificial data, which allowed us to study different execution plans in a controlled way. On the desktop computer, our test databases contained between 59.7 and 149.7 million triples, described 1 million resources, and occupied 6.6 to 15 GB of space. On the mobile device, the databases contained between 4.55 and 10.5 million triples, described 100 000 resources, and required between 414 MB and 1 GB of storage. Due to the lower memory bandwidth of the mobile device, we reduced the database page size from 16 KB (as used on the desktop computer) to 2 KB.

We ran all test queries ten times consecutively on warm caches and measured the execution times. Our figures report the median of the ten test runs. In addition, we counted the logical page requests to get an impression on the main memory (if not disk) I/O behavior of the execution plans. Moreover, to examine only the physical database operators, we measured execution times and page requests both including and excluding the dictionary. I. e. in the latter case, the raw internal IDs and addresses were returned instead of resolving the external string representation of the values.

We ran all tests for the desktop computer on a *Dell Optiplex 755* equipped with an Intel Core2 Quad Q9300 CPU running at 2.50 GHz and 4 GB of main memory. We used two striped 250 GB SATA 3.0 GB/s hard drives spinning at 7.200 RPM. The test machine ran a 64 bit 2.6.31 Linux kernel.

The tests on a mobile device were run on a *Nokia N900* smartphone that is equipped with a Texas Instruments OMAP 3430 SoC ARM Cortex-A8 processor running at 600 MHz. It possesses 256 MB of Mobile DDR main memory and 768 MB of flash-based swap space for a total of 1 GB virtual memory. Its persistent storage consists of 32 GB built-in eMMC flash memory and a 256 MB NAND flash chip that accommodates the operating system.

3.5.3 Resources versus Attributes

The most important and most interesting question is, how the different execution plans behave for different amounts of queried resources $|R|$ and retrieved attributes $|A|$. To answer this, we ran a large series of queries on a database containing 1 million resources. Every resource carried exactly 30 attributes with random values, so that they can be assumed distinct. To simulate resource identification, we created 1000 resource groups of different sizes. The group members were chosen randomly. A resource is a member of a particular group, if it is connected to the group URI via the `inResourceGroup` predicate. This enabled us to select a defined amount of resources using a single triple pattern.

The test queries retrieved between $|A| = 2$ and $|A| = 30$ attributes. Figure 3.8 shows the results on the desktop computer with $|R| = 100$, 2500, and 5000 resources. Figure 3.9 shows the results on the mobile device for $|R| = 10$, 250, and 500 resources. In both cases, we measured the execution times and logical page requests both retrieving only the *raw* internal IDs, and the *total* effort including the dictionary overhead. The results lead to the following observations:

1. The raw figures of our approach both on the SPO index and on the attribute retrieval index are always nearly horizontal lines. Thus, for a given number of resources $|R|$, our approach practically requires constant time and page accesses, independent of $|A|$. This is because it accesses all attributes of a resource from a contiguous memory area, so that it effectively makes no difference, how many attributes it retrieves. The canonical plans scale linearly with $|A|$, as they require one join per retrieved attribute.

2. The difference between the raw and the total figures, i. e. the dictionary overhead, is the same for canonical plans and our approach on the SPO index. On the attribute retrieval index, the dictionary overhead is about 25% less for execution times and nearly 50% for page accesses. This is because it does not need to resolve the internal IDs of the attribute values.

3. The attribute retrieval index shows far better execution times than all other approaches, both with and without the dictionary overhead. However, it accesses the identical amount of pages as our approach on the SPO index. Thus, the leaf page layout must make the difference. The attribute retrieval index only accesses a small part of every leaf page and only uncompresses what it needs to read. The other indexes must load and uncompress the entire page first.

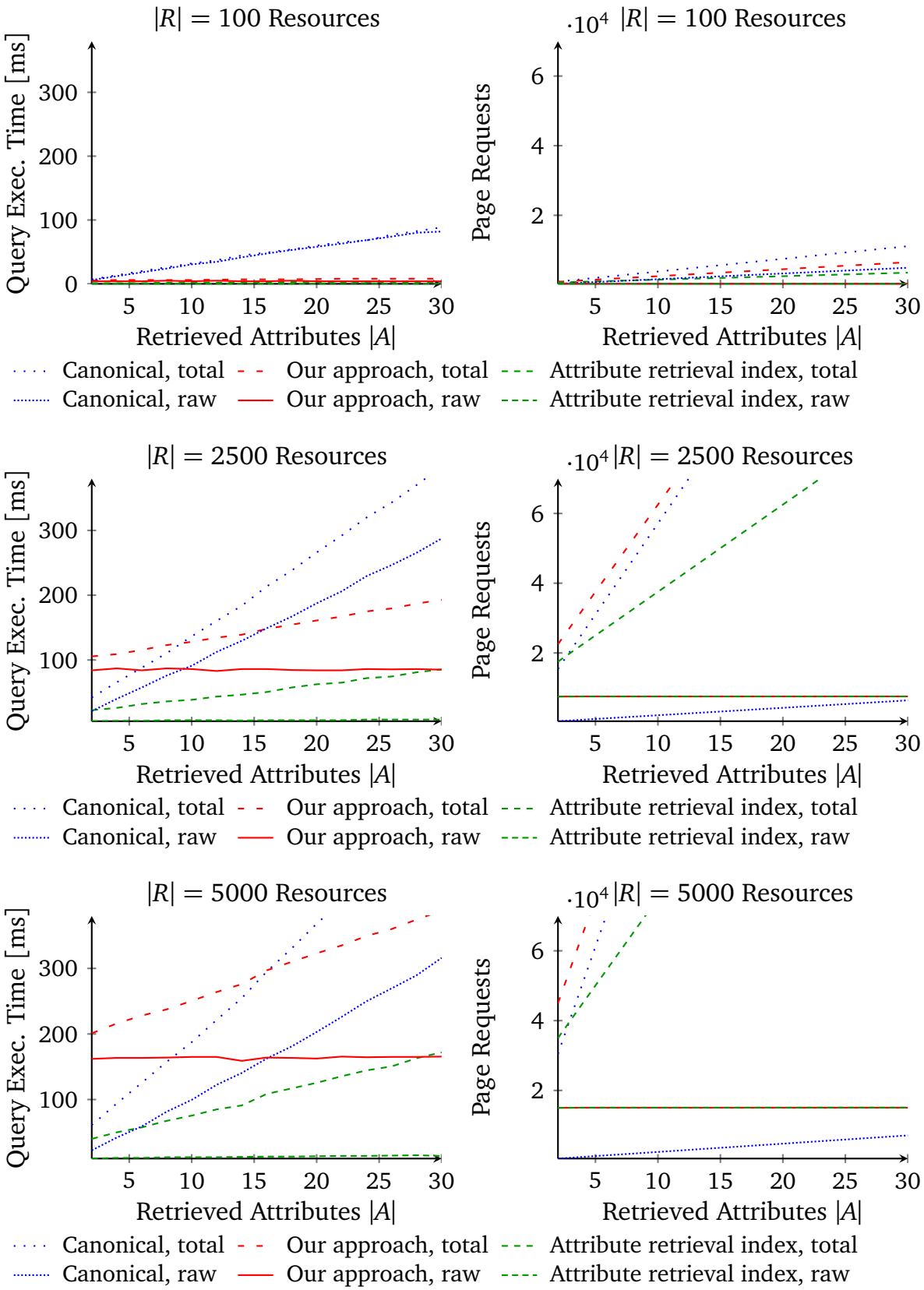


Figure 3.8: Resources versus Attributes: Query Execution Time and Page Requests, measured on the **Desktop Computer**

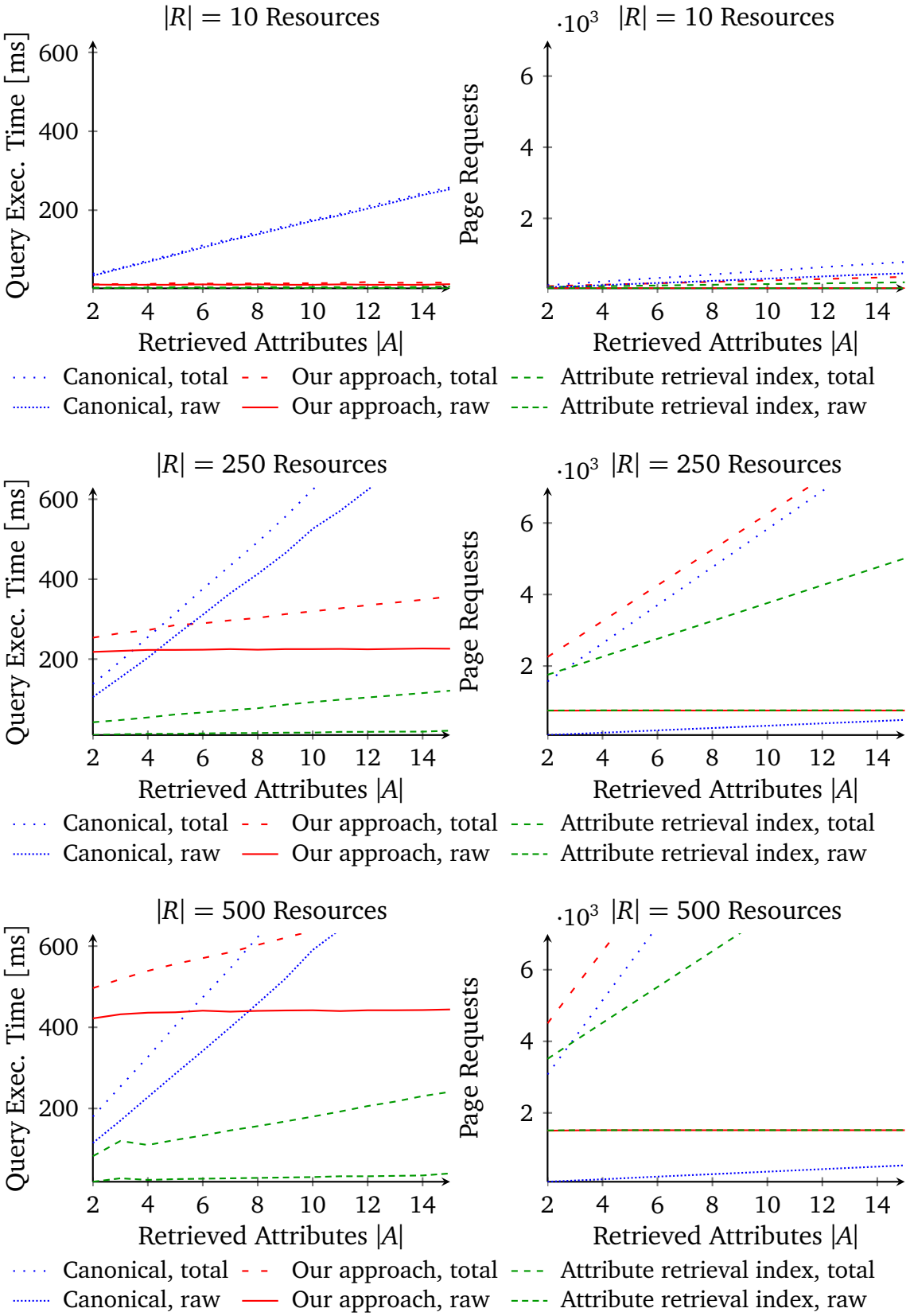


Figure 3.9: Resources versus Attributes: Query Execution Time and Page Requests, measured on the **Mobile Device**

4. The break-even-point θ of canonical plans and our approach on the SPO index is (in this evaluation) for the desktop computer and the mobile device around

$$\theta_{desktop} = \frac{|R|}{|A|} \approx 330 \text{ and } \theta_{mobile} = \frac{|R|}{|A|} \approx 80,$$

respectively. Higher values favor canonical plans, below our approach on the SPO index is faster. The pivot index scans on the attribute retrieval index were always the fastest.

5. Using our approach on the attribute retrieval index, typical amounts of resources and attributes handled by a mobile application can be retrieved on the mobile device in 200 ms, which is suitable for an interactive application.

3.5.4 Multi-Attributes

As pointed out in Section 3.2.3, SPARQL returns the cross-product over multi-attributes. This results in exponentially growing results, if the number of retrieved attributes $|A|$ is increased. We ran large test series on databases similar to the one described in Section 3.5.3, except that every resource carried each attribute two, three, or four times, respectively.

For lower amounts of computed cross-products, the results were similar to those of Section 3.5.3. The more cross-products the multi-attributes caused, the closer the raw execution times of the different plans became (i. e. not counting dictionary overhead). For extremely high numbers of result tuples, the canonical plan showed faster raw execution times on the desktop computer, even though it accessed significantly more pages. We presume that this is due to the more sequential memory access pattern of the streamlined merge joins that leads to better CPU cache-hit rates across the triples of different resources when computing the cross-products. A pivot index scan bears excellent memory locality for one resource, but it causes random memory access for different resources.

Figure 3.10 shows the raw execution times of one test series on a database with two multi-attributes and one series with three multi-attributes. For 1000 resources, three attributes were retrieved once and a fourth attribute was selected one to ten times. It is observable, that the execution times become similar with high amounts of multiply-selected attributes. For lower amounts, however, our approach is more than an order of magnitude faster on the attribute retrieval index. The results are roughly comparable to a different test series on the same database that retrieved the same amount of distinct attributes, i. e. without multiply selected attributes. Thus,

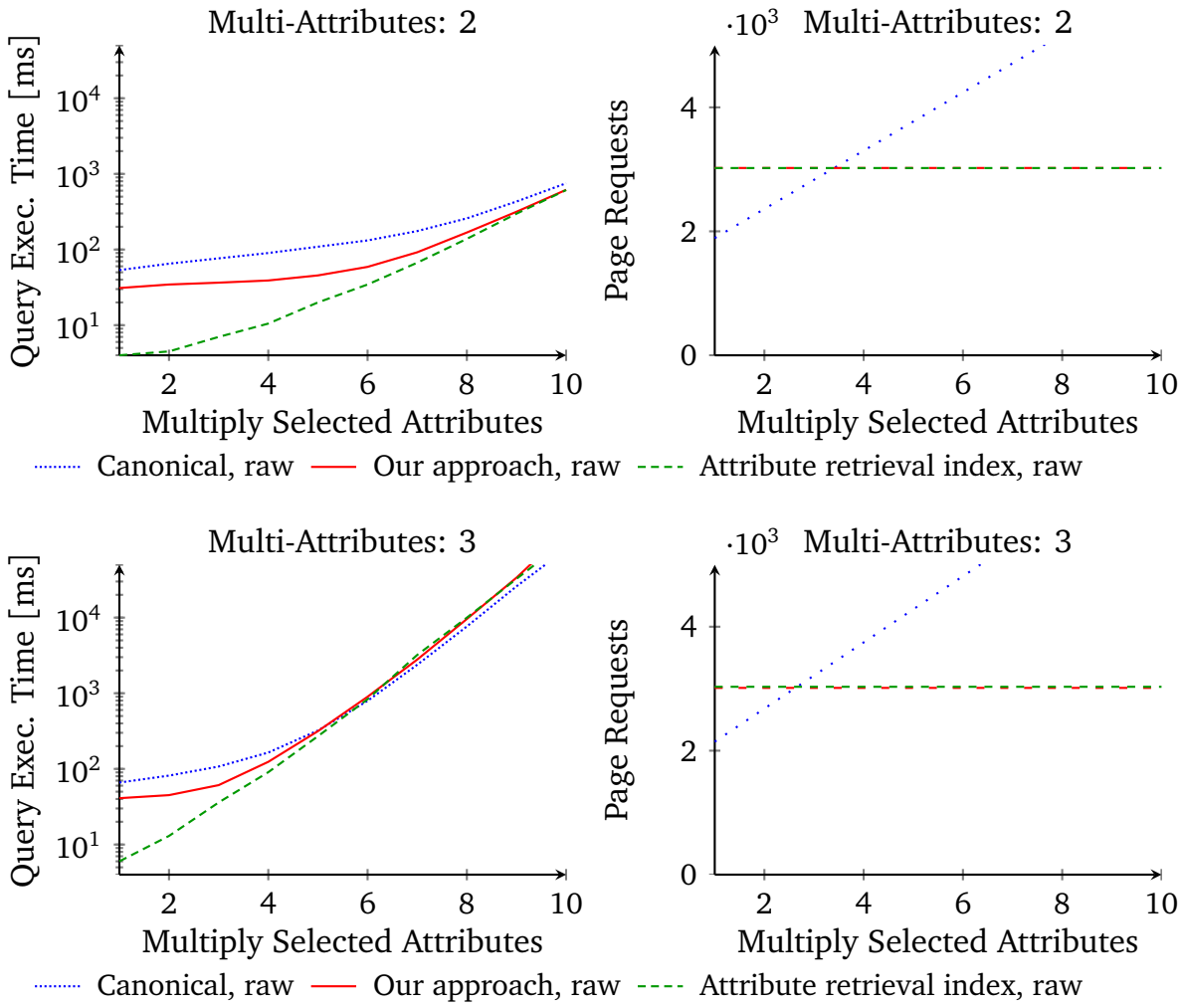


Figure 3.10: Multiply selected attributes, on databases with 2 or 3 multi-attributes, $|R| = 1000$ resources, measured on the **Desktop Computer**

multiply selected attributes are hardly different from distinct selections, as long as the number of cross-products is the same.

Figure 3.11, similarly, shows the raw execution times on the mobile device for 100 resources on databases with two and three multi-attributes. The results are comparable, except that our approach (both on the SPO index and on the attribute retrieval index) is always faster than the canonical plan, even for very large amounts of computed cross-products.

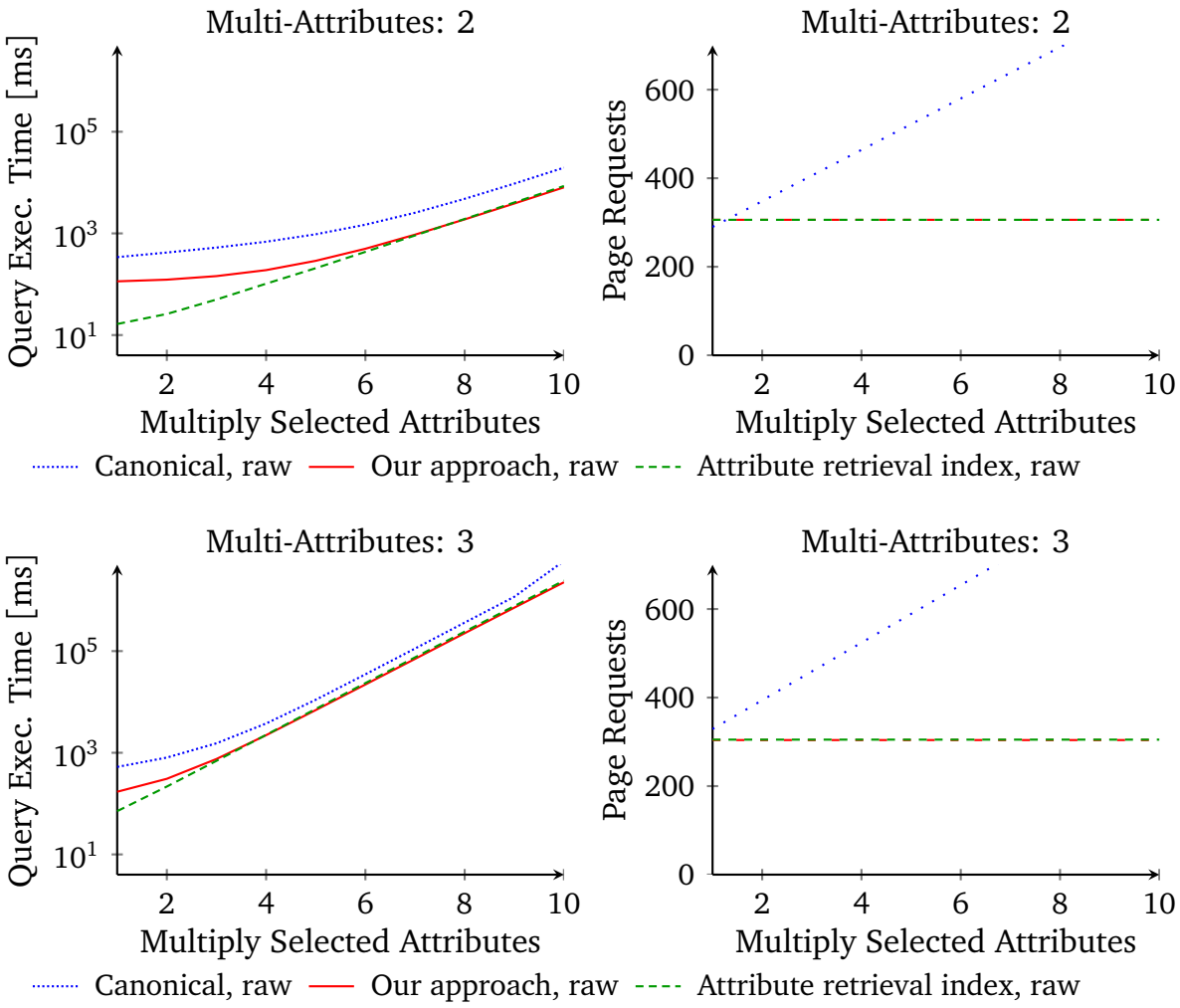


Figure 3.11: Multiply selected attributes, on databases with 2 or 3 multi-attributes, $|R| = 100$ resources, measured on the **Mobile Device**

3.5.5 Selective Attributes

As discussed in Section 3.3.5, naively applying our approach on attributes which only occur at a small fraction of resources will result in many failing index lookups. To measure this effect, we used the database of Section 3.5.3 and added 100 *selective* attributes, which occurred only at 0.1% to 10% of all resources. We randomly chose the resources which carried a selective attribute. Yet we ensured that the fraction of resources with selective attributes was the same for all resource groups. Also, for every selective attribute, every group contained at least one resource that carried the attribute.

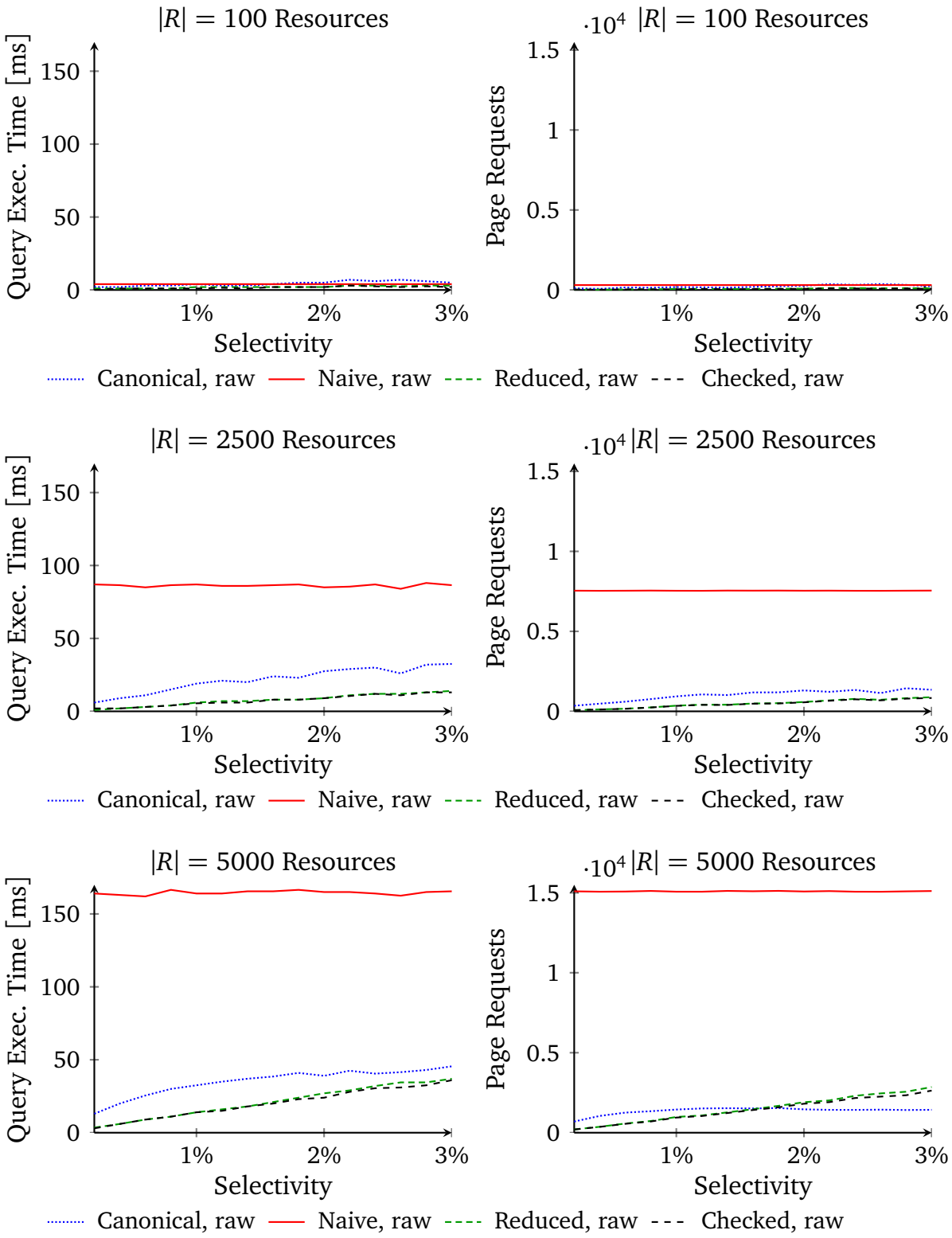


Figure 3.12: Selective attributes: Query execution time and page requests for $|R| = 100$, 2500, and 5000 resources, measured on the **Desktop Computer**

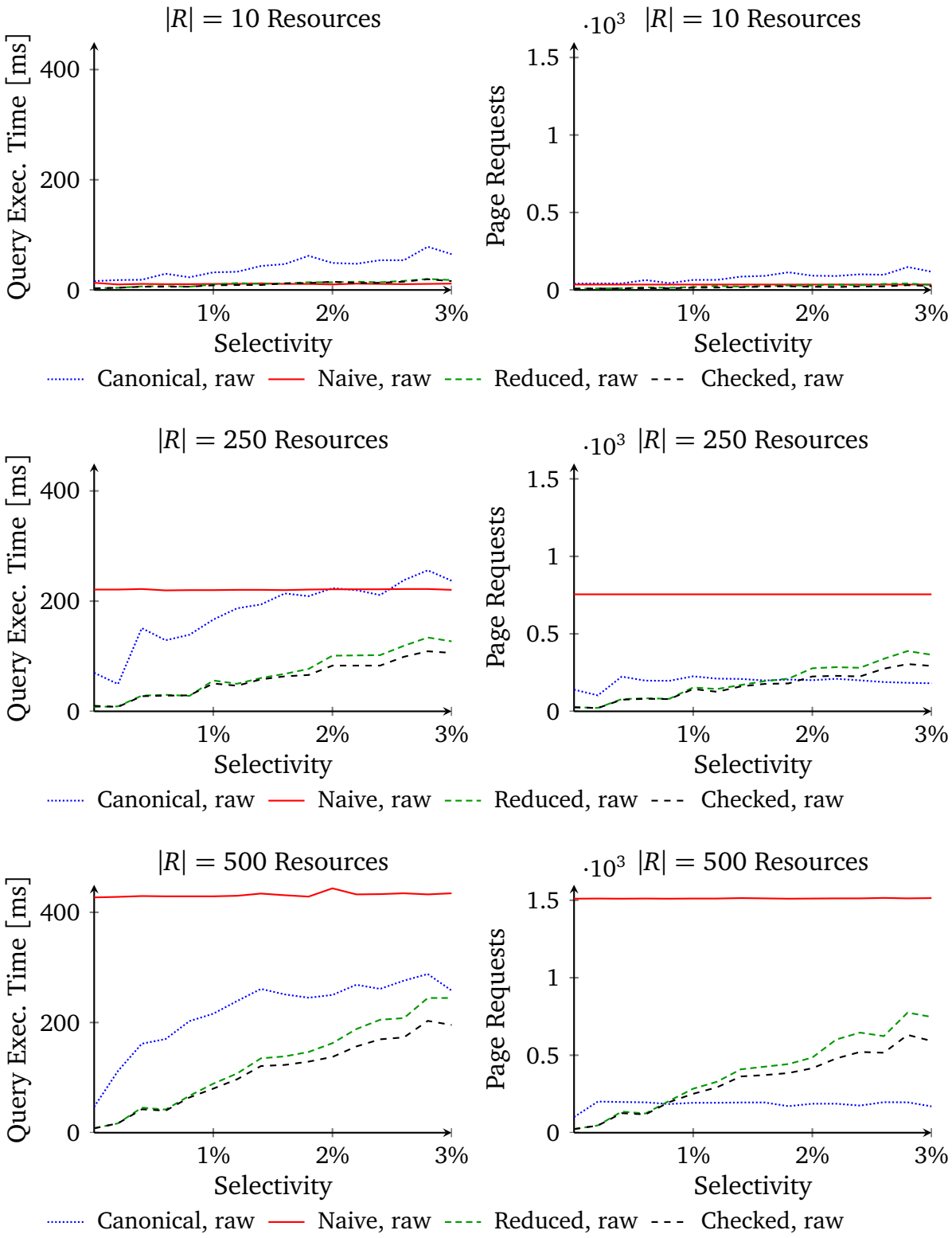


Figure 3.13: Selective attributes: Query execution time and page requests for $|R| = 10$, 250, and 500 resources, measured on the **Mobile Device**

The test queries retrieved four "ordinary" attributes, as in Section 3.5.3, and one selective attribute from different amounts of resources. Following our results from Section 3.5.3, we would expect our approach to be superior up to about 1650 resources on the desktop computer and up to 400 on the mobile device, if all attributes were always present. We compared a canonical execution plan, a naive application of our approach, the "reduced" approach of Section 3.3.5, and the "checked" approach of Section 3.3.5.

Figure 3.12 shows the measured execution times and page requests for $|R| = 100$, 2500 and 5000 resources measured on the desktop computer. As expected, the naive application of our approach is by far worse than the canonical plan, both for execution times and page requests. The "reduced" and "checked" plans, on the other hand, are more than twice as fast as the canonical plan and also request fewer pages, except for less selective cases on 5000 resources. The "checked" plan is always slightly better than the "reduced" plan.

Likewise, Figure 3.13 shows the measured execution times and page requests for $|R| = 10$, 250 and 500 resources measured on the mobile device. The big picture is about the same as on the desktop computer; naively applying our approach is far worse than the canonical plan. Also, the "reduced" and "checked" plans are again significantly faster than the canonical plan. The canonical plan still often causes more page requests than the "reduced" and "checked" plans, but it reaches the break-even-point much sooner as on the desktop computer. Also, the advantage of the "checked" plan over the "reduced" plan is greater than on the desktop computer. Both observations can be explained by the smaller page size which we used on the mobile device: Smaller pages cause a smaller fan-out of inner B-tree nodes, which increases tree height and thus the page requests required for index lookups. The canonical plan requires a lot less index lookups (one per triple pattern), as it reads triples from contiguous index ranges. Contrarily, a pivot index scan executes one index lookup per resource. The "checked" plan uses the aggregated index to identify the resources which carry selective attributes. Due to the smaller page size, B^+-tree height is more of an issue, so the 25 % higher fan-out creates even larger advantages compared to the full triple index, which the "reduced" plan uses.

3.6 Summary and Outlook

The W3C Resource Description Framework (RDF) supports managing semi-structured data without a predefined database schema. RDF models data as triples which may represent a relation between two resources or an attribute of a resource. So far, the focus of most RDF query processors has been on finding complex graph patterns

in RDF data based on the relations between resources. This typically involves a high number of joins. By contrast, obtaining a record-like view on the attributes of resources, as natively supported by record-based DBMS, imposes unnecessary performance burdens in these query processing models. They must join the individual attributes to assemble the final result records, as they do not differ between finding resources that match a graph pattern and retrieving resource attributes.

We proposed a processing model that splits queries to RDF data into two conceptual parts: *resource identification* and *attribute retrieval*. First, resource identification determines the resources of interest for the query. In a second step, attribute retrieval fetches the queried attributes in a single step for each identified resource. For this, we exploit an index that stores all attributes of a resource contiguously, which saves a large number of joins. Most RDF stores already possess such an index anyway; they just do not exploit it in our proposed way. In addition to that, we proposed an index structure that is specifically designed for the access pattern of attribute retrieval. Our performance evaluation showed that our processing model is clearly superior for larger numbers of retrieved attributes and moderately large amounts of resources. This holds on desktop computers as well as on mobile devices. Moreover, our attribute retrieval index further improves performance by large margins. Most notably, using our approach on the attribute retrieval index, typical amounts of resources and attributes handled by a mobile application can be retrieved in times that are suitable for an interactive application.

Future Work

The underlying concept of our presented query processing model is not restricted to triple stores on a mobile device. It applies to RDF data management in general. Furthermore, we believe that it is applicable on any decomposed storage model. It is an interesting challenge to apply it on other types of database systems, such as column-oriented databases or object databases, in the future.

The next chapter addresses another central aspect of the Data Management Layer in our architecture: the integration and management of spatial data in RDF triple stores.

4

Deep Integration of Spatial Query Processing into RDF Triple Stores

A central requirement of our architecture is interoperability with spatial data, as discussed in Section 1.1.2. In contrast to most other data management scenarios, on a mobile device a large share of the managed data possesses spatial relevance. Also, mobile devices accompany the user most of the time, so that also non-physical entities, such as text messages, may obtain spatial references indirectly through the position of the user at a given time. Thus, spatial references create a general join criterion for most resources. However, to achieve interoperability, these joins should be executed in the Data Management Layer and not in hard-wired application code. Thus, spatial query processing functionality must be deeply integrated into the data management system, as illustrated in our system architecture in Figure 2.2.

This chapter presents an approach to integrate spatial query processing deeply into the RDF data model and into RDF triple stores, as published in [Brodt et al., 2010a]. I. e., we enable RDF triple stores to process spatial queries and analyses efficiently. This is a leap forward from the state of the art (Section 4.1): For the RDF data model, deep integration means that we model geographic data in RDF as complex objects represented as literals of an abstract geometry type (following the OpenGIS Simple Features Specification [Herring, 2006]). By this, spatial features, such as coordinate-based points, line strings or polygons, are treated as data types like strings or numbers, and can be manipulated, queried and processed by a standardized set of spatial functions (Section 4.2). Likewise, we propose to integrate spatial predicates

by means of SPARQL filter functions on this geometry type (Section 4.2.2). This has the advantage of being expressible in W3C's SPARQL query language without any language extensions. To deeply integrate these concepts into RDF query processing, we consider two approaches, a spatial selection operator and a spatial index, which we implemented in a native RDF triple store (Section 4.3) and evaluated using generated and real-world spatial RDF data (Section 4.4). Furthermore, we address cardinality estimation for SPARQL queries containing spatial query predicates, so that a query optimizer is capable of choosing optimal execution plans (Section 4.5). We conclude the chapter with promising future research directions in Section 4.6.

4.1 State of the Art and Foundations

In this section, we review existing approaches to spatial RDF modeling and processing, and give foundations about the underlying technology when needed to understand our approach. In the illustrating examples throughout the chapter, we use the namespaces defined in Table 4.1.

4.1.1 RDF Data Management

As explained in Section 3.1.1, the characteristics of RDF facilitate representing, exchanging, combining, and linking information from different, heterogeneous data

Prefix	Namespace URI and Comment
:	`http://example.org/application-specific-ontology` Default namespace as an example for an application-specific ontology
geo:	`http://www.w3.org/2003/01/geo/wgs84_pos` The W3C Geo specification [Brickley, 2003]
georss:	`http://www.georss.org/georss` The GeoRSS standard [Singh et al., 2009]
geordf:	`http://example.org/geo` Namespace of our approach
gml:	`http://www.opengis.net/gml` The OpenGIS Geography Markup Language (GML) Standard [OpenGIS, 2000]
osm:	`http://example.org/osm` The test data used in our evaluation in Section 4.4

Table 4.1: Namespace prefixes used in this chapter

sets. Huge repositories exist that publish and link data sets in RDF format, e. g., the data sets on Linked Data,[1] the Uniprot data set,[2] or many data sets published on the data.gov catalog[3] of the USA. Many of these data sets contain geographical or symbolic location information, and spatial relations between data entities play a crucial role for searching and analyzing spatial data. Consequently, the ontological modeling of geographic entities and their geospatial relations is a significant research direction in the Geographic Information Science (GIS) community. As mentioned before, RDF treats relationships as first class objects, which suits it very well to model and query complex relationships between resources. In addition, Semantic Web technologies, to which RDF belongs, deal well with problems such as non-unique names or subclass relationships, which typically occur in data integration tasks that are very common when working with spatial data [Kammersell and Dean, 2006]. Moreover, RDF provides schema flexibility which is useful for analyzing and integrating poorly structured data, e. g., web- or community-based data, such as map data from the OpenStreetMap (OSM) project [Auer et al., 2009].

We introduced RDF triple stores in Section 3.1.3, which are capable of processing very large amounts of RDF data efficiently. As long as primitive data types, most notably strings, are concerned, these systems can be well utilized to analyze geospatial data—depending on the used location model [Bauer et al., 2002]: systems working with symbolic coordinate systems, for instance many indoor location systems, typically use a graph model connecting symbolic names. These systems can directly make use of efficient RDF data management. However, queries involving geographical positions represented as coordinates, e. g., range queries or nearest-neighbor queries, require vector-based coordinates. If these coordinates are represented as a string, a DBMS cannot support the specific characteristics of location information (e. g., multi-dimensional data and the need for spatial indexes) or provide type-safe spatial functions. However, a widely accepted standard for representing spatial information exists [Herring, 2006], and the RDF data model allows for extension by new data types. But RDF triple stores do usually not address native spatial data processing yet. Until now, efficient spatial analyses of RDF data would typically require external processing, e. g., using a geo-enabled database.

Modeling Spatial Features in RDF

There are many ways to model spatial features in RDF. Spatial features, as standardized by the OpenGIS Simple Features Specification [Herring, 2006], are complex

[1] `http://linkeddata.org/`
[2] `http://dev.isb-sib.ch/projects/uniprot-RDF/`
[3] `http://www.data.gov`

```
:Photo123  geo:lat  "48.77".
:Photo123  geo:lon  "9.18".
```

Listing 4.1: W3C Geo example

structures. A point, for instance, consists of two or three coordinates, a linestring or a linear ring comprises many points and a polygon is bounded by a linear ring; it may also have holes, which are linear rings, too. These relationships can be directly modeled as RDF triples. This approach decomposes every spatial feature into several separate triples resulting in a large total amount of triples for spatial data. For feature types consisting of multiple parts, such as Polygon or MultiLineString, every part is modeled as a separate data object having its own URI. This is useful if other data objects need to reference a single part of a spatial feature. E. g., the boundary of a lake may be explicitly referenced to represent a hole in the surrounding meadow. However, the decomposed approach is unfavorable for processing the feature as a whole, e. g., perform calculations, index it, etc., as the feature first needs to be reassembled from its parts.

The W3C Geo Vocabulary [Brickley, 2003] (now considered deprecated) was an extreme example of the decomposed approach. It represented points using two separate triples for latitude and longitude. It did not support any other feature types, though. For instance, to express the geographic position where a photo was taken, W3C Geo would add the two triples shown in Listing 4.1 to the RDF graph.

The initial version of GML had an explicit RDF/XML binding [OpenGIS, 2000]. Later GML versions use an object-property pattern and XML linking which can be directly mapped to RDF. Thus, it is straightforward to transform GML data to an RDF graph. The direct translation results in the decomposed approach with the difference that a list of coordinates is modeled as a single string containing space-separated floating point numbers.

GeoRSS GML [Singh et al., 2009] uses a GML profile to embed spatial data into RSS feeds, which can be represented in RDF. GeoRSS GML represents spatial features in a class hierarchy which consists of the abstract Geometry class and its subclasses Point, Line, Box, and Polygon. GeoRSS GML does not support multi-geometries or polygons with holes. Every spatial feature is represented as a resource having its own URI. In the preferred serialization format, the coordinates are supplied as a single string literal containing a space-separated coordinate list. This string literal is connected to the spatial feature via the `gml:pos` predicate. GeoRSS GML models the location of the photo from the upper example as shown in Listing 4.2.

As can be seen, GeoRSS GML uses two RDF triples to specify the location: one to specify the class of the feature and one to supply the coordinates. I. e., a more

```
:Photo123  georss:where  :Point209.
:Point209  a             gml:Point.
:Point209  gml:pos       "48.77 9.18".
```

Listing 4.2: GeoRSS GML example

complex feature type, e. g. a polygon, would require two triples as well. Thus, GeoRSS GML can be seen as a *hybrid* approach which does model a spatial feature as a discrete resource, but does not model its parts separately. The feature still carries its own URI, which enables adding further metadata to it, e. g., accuracy or provenance information. It is also reusable, as other resources may reference it. Yet, to process the feature, the two RDF triples still need to be joined.

4.1.2 The SPARQL Query Language

Section 3.1.2 introduced the SPARQL query language, which was standardized by W3C [Prud'hommeaux and Seaborne, 2008] to search RDF repositories. SPARQL expresses graph pattern matching queries on an RDF graph as conjunctions (and also disjunctions) of triple patterns which define variable bindings. In addition to the pattern-based search, these variable bindings can be restricted through filters which allow selections on the *values* of the variable bindings.

Technically, SPARQL filters are functions returning a boolean. SPARQL specifies a number of built-in filter functions, such as `regex` or the usual comparison operators (`>`, `!=`, ...). Yet, SPARQL explicitly allows additional filter functions that are identified by a URI. Listing 4.3 shows an exemplary SPARQL query that returns the title and file name of all photos whose title contains the substring ''`Harry`''.

Expressing Spatial Query Predicates in SPARQL

Kolas [2008] proposed to express spatial query predicates using so-called *query premises*, which declare the query parameters as a non-materialized part of the graph pattern in SPARQL. Kolas gives an example similar to the query depicted in

```
SELECT ?title ?file WHERE {
   ?photo  a          :Image.
   ?photo  :fileName  ?file.
   ?photo  :title     ?title.
    FILTER regex (?title, ".*Harry.*")
}
```

Listing 4.3: SPARQL example using a filter expression

Listing 4.4, which is meant to find the file names of all photos within distance 1 from point `(48.765 9.175)`.

This solution mixes graph patterns on the materialized RDF data set on one hand and spatial query parameters on the other hand. The two parts are connected via the `rcc:part` predicate, which expresses the spatial relationship to be evaluated on the fly (Kolas also supports `rcc:connected`). This complicates the query processor and is difficult for humans to read, as different parts of the query have different semantics. To address this problem, Kolas suggests to move the query premise to a separate section of the query and thus proposes the scheme `SELECT ?x PREMISE {...} WHERE {...}`. However, this does not comply with the SPARQL specification.

To express query predicates which require parameters, SPARQL provides filter functions. Perry [2008] formulates spatial query predicates using a `SPATIAL FILTER` clause followed by calls to spatial comparison functions. This is similar to SPARQL filter functions, but does not match the exact syntax (due to further extensions, this was not Perry's goal). Kolas criticizes that expressing spatial relations as functions breaks the RDF and SPARQL philosophy of modeling all relations between objects as graph edges [Kolas, 2008]. Yet, filter functions appear a more natural way to formulate the spatial query predicates, as done by relational spatial databases.

Spatial RDF Databases

Perry [2008; 2006] presented a framework for analysis of spatial and temporal RDF data that was implemented as a set of user-defined functions in Oracle DBMS. It models geographic features in an ontology based on GeoRSS GML [Singh et al.,

```
SELECT ?file WHERE {
   ?photo     a                    :Image.
   ?photo     :fileName            ?file.
   ?photo     georss:where         ?location.
   ?location  rcc:part             ?buffer.
   ?buffer    a                    gml:Buffer.
   ?buffer    gml:radius           1.
   ?buffer    gml:bufferGeometry   ?point.
   ?point     a                    gml:Point.
   ?point     gml:pos              "48.765 9.175".
}
```

Listing 4.4: SPARQL example using a *query premise* to express spatial query predicates (adapted from [Kolas, 2008])

2009], but stores them as complex objects in relational tables. Its major disadvantage is the lack of a standardized query language.

Kolas and Self [2007] proposed to use W3C's SPARQL query language [Prud'hommeaux and Seaborne, 2008] for spatial RDF data. They use the GeoRSS RDF vocabulary to model spatial features and formulate spatial queries by means of the query premises, as discussed above. Their implementation [Kolas, 2008] builds on top of the Jena Semantic Web Framework [Jena] and uses a main memory grid file. No performance results were reported.

Recently, Virtuoso implemented support for spatial data[4] using an approach similar to ours. They support spatial joins (which we will address in future work), but are restricted to point data thus lacking arbitrary shapes, e. g. multi-polygons. No performance results were reported on spatial queries.

To the best of our knowledge, no approach exists that natively integrates arbitrary geographic information into the RDF data model and allows efficient processing of spatial operators using a standardized query language. Thus, we propose our approach of *deep integration*: location information is treated like other basic data types (e. g. string), which all benefit from type-safe type-specific functions, query predicates, and efficient processing due to type-specific index support of industrial-strength data management systems.

4.2 Modeling and Querying Spatial Literals in RDF

This section introduces our approach for deep integration of spatial features into RDF processing. For this, we have to extend both the modeling of spatial features in RDF triples, and provide spatial predicates in the standard RDF query language, SPARQL. However, our approach fully complies with the RDF and SPARQL specifications.

4.2.1 Spatial Literals in RDF

As described in Section 4.1.1, most existing approaches decompose the spatial information into multiple RDF triples. Our approach to model spatial features in RDF is entirely opposed to the decomposed modeling approach. We represent spatial features as a complex self-contained data type and store them in RDF literals. The literals contain the spatial features expressed in the Well-know Text (WKT) format, as standardized in the OpenGIS Simple Features Specification [Herring, 2006]. The literals carry a type URI which indicates that the literal must be processed as a

[4] http://docs.openlinksw.com/virtuoso/rdfsparqlgeospat.html

```
:Photo123  :takenAt  "POINT (48.77 9.18)"^^geordf:geography.
```

Listing 4.5: A spatial feature expressed as a typed literal in RDF

spatial feature rather than as an ordinary string. Listing 4.5 shows how our approach represents the position at which the exemplary photo was taken.

Syntactically, any predicate can connect the literal to any resource. Thus, to process the spatial feature, only the single RDF triple which contains the spatial literal is required, which has two significant advantages:

1. The spatial feature can be processed independently of a particular schema or ontology. In GeoRSS GML, by contrast, the processing system must *know* that the `gml:Point` type and the `gml:pos` predicate are related to spatial data to interpret them accordingly. Data using a different schema will not be supported unless the processing system is told the spatial RDF classes and predicates.

2. The processing system is simplified, as it does not need to reassemble the spatial feature from a number of triples. It may process the RDF data triple by triple.

Our approach does not assign a URI to a spatial feature, so that it cannot be directly referenced or augmented by metadata. However, if this is required, one can easily introduce *place resources* in a specific ontology. These place resources, naturally, are identified by their URI, carry the geometry literal, and may possess further metadata. Their URIs may even be used as symbolic coordinates. Still, the place resources would keep all information related to geographic coordinates in the single RDF triple which carries the spatial literal. All further triples related to the place resource are "ordinary" RDF and can be designed in any ontology. The address of Harry's house in Figure 3.1 is very similar to such a place resource. Listing 4.6 shows the photo example using a place resource.

4.2.2 SPARQL Filter Functions

Clearly, it is desirable to express spatial queries on RDF data in SPARQL as well, rather than introducing yet another specialized query language. From this arises a

```
:Photo123  :takenAt     :Place927.
:Place927  :source      "GPS sensor 4711".
:Place927  :accuracy    "19.56 m".
:Place927  :satellites  6.
:Place927  :coords      "POINT(48.77 9.18)"^^geordf:geography.
```

Listing 4.6: A spatial feature represented by a *place resource*

```
SELECT ?title WHERE {
    ?photo  :title    ?title.
    ?photo  :takenAt  ?geo.
     FILTER geordf:within (?geo,
        "POLYGON((48.775 9.175, 48.775 9.185, ...))"^^geordf:geography)
}
```

Listing 4.7: A spatial query predicate expressed as a SPARQL filter function

challenge, as SPARQL is designed to search for exact patterns in a (materialized) RDF graph. For spatial joins on some spatial predicates, such as *within*, *covers*, *crosses*, etc., it would be possible to materialize the respective relationships as an explicit RDF triple (which would require the spatial features to carry a URI). These triples could be queried by ordinary SPARQL graph patterns, but would lead to a combinatorial explosion in the total amount of RDF triples. Generally, spatial predicates rarely search for exact relations between data objects but involve calculations which often require parameters. Examples include the maximal distance in a range query or a given constant geometry with which to compare the query result.

Our approach to express spatial query predicates in SPARQL uses filter functions that are identified by a URI. We use the functions of the OpenGIS Simple Features Specification [Herring, 2006], as they are well-established. We defined a URI for each of them. The filter functions act on variables which bind a spatially typed literal, as described in Section 4.2. Thus, the filter functions complement well our approach to model spatial features in RDF as typed literals. Geometry constants to compare the value of the variable are specified as spatially typed literals, too; they are given in the Well-know Text (WKT) format and carry a URI denoting the spatial type. Listing 4.7 shows a fully standard-compliant SPARQL query to find all pictures that were taken within a given area that specifies the spatial query predicate as a filter function.

4.3 Implementation

We consider two fundamental approaches to implement an RDF triple store with support for spatial query processing: a spatial selection operator and a spatial index. In a full-fledged DBMS, both may be combined for optimal performance.

A spatial selection operator can be implemented on top of an existing triple store. Thus, the query is split into a pure RDF pattern matching query and the spatial query predicate. First, the pattern matching query is evaluated entirely by the triple store. In a second step, the selection operator evaluates the spatial query predicate on every tuple returned by the triple store and discards tuples that do not match. The two

steps are performed sequentially (at least conceptually) and there is no way for the selection operator to restrict the query result at an earlier stage. To avoid unnecessary intermediate results, the selection can be pushed down in the query graph. However, this rules out an implementation on top of the triple store, as the query optimizer of the triple store needs to be modified. Also, the selection operator must cope with the internal data representation of the triple store in that case.

A spatial index, on the other hand, may select only spatial features which match the spatial query predicates, right from the start. A spatial index cannot be deployed without a comprehensive deep integration of spatial query processing functionality into the triple store. First of all, the spatial index requires its own database segment to store the spatial features. Whenever spatial data is loaded, the triple store must recognize it and forward it to the spatial index. Moreover, the query optimizer must be deeply modified to recognize the spatial index and optimize the joins of the spatial features it returns with other intermediate query results. Naturally, the spatial index must also provide the spatial features in a way they can be joined, i. e., it must follow the internal processing mechanisms of the triple store.

We implemented both approaches. As in Chapter 3, we used RDF-3X [Neumann and Weikum, 2008] version 0.3.4 as the starting point. However, as all state-of-the-art triple stores share common characteristics (see Section 3.1.3), our results are applicable for other triple store implementations as well.

4.3.1 Architecture and Processing Model

To illustrate our implementation, we first introduce the architecture and the processing model, as shown in Figure 4.1. The triple store consists of a query front end, a query optimizer, physical query operators, a dictionary, and indexes. As explained in Section 3.1.3, RDF-3X indexes the RDF triples (*S*ubject, *P*redicate, *O*bject) in all six possible permutations: SPO, SOP, PSO, POS, OSP, OPS. The indexes do not list the actual triples but consist of integer IDs, which the dictionary maps to the respective URIs or literals. This saves memory and enables fast join processing.

When the query front end receives a SPARQL query (1), it parses the query and immediately calls the dictionary (2) to resolve all URIs and literals to integer IDs. The logical query graph, which the semantic analysis produces, is unaware of URIs or literals but uses these IDs exclusively. Subsequently, the query optimizer finds an optimal execution plan (3), as described in Section 3.3. The resulting physical operator graph is instantiated using the query operators (4). The operators query the indexes (5) and determine the query result. As all internal processing is done using IDs, the dictionary finally needs to map the query result back to URIs and literals (6).

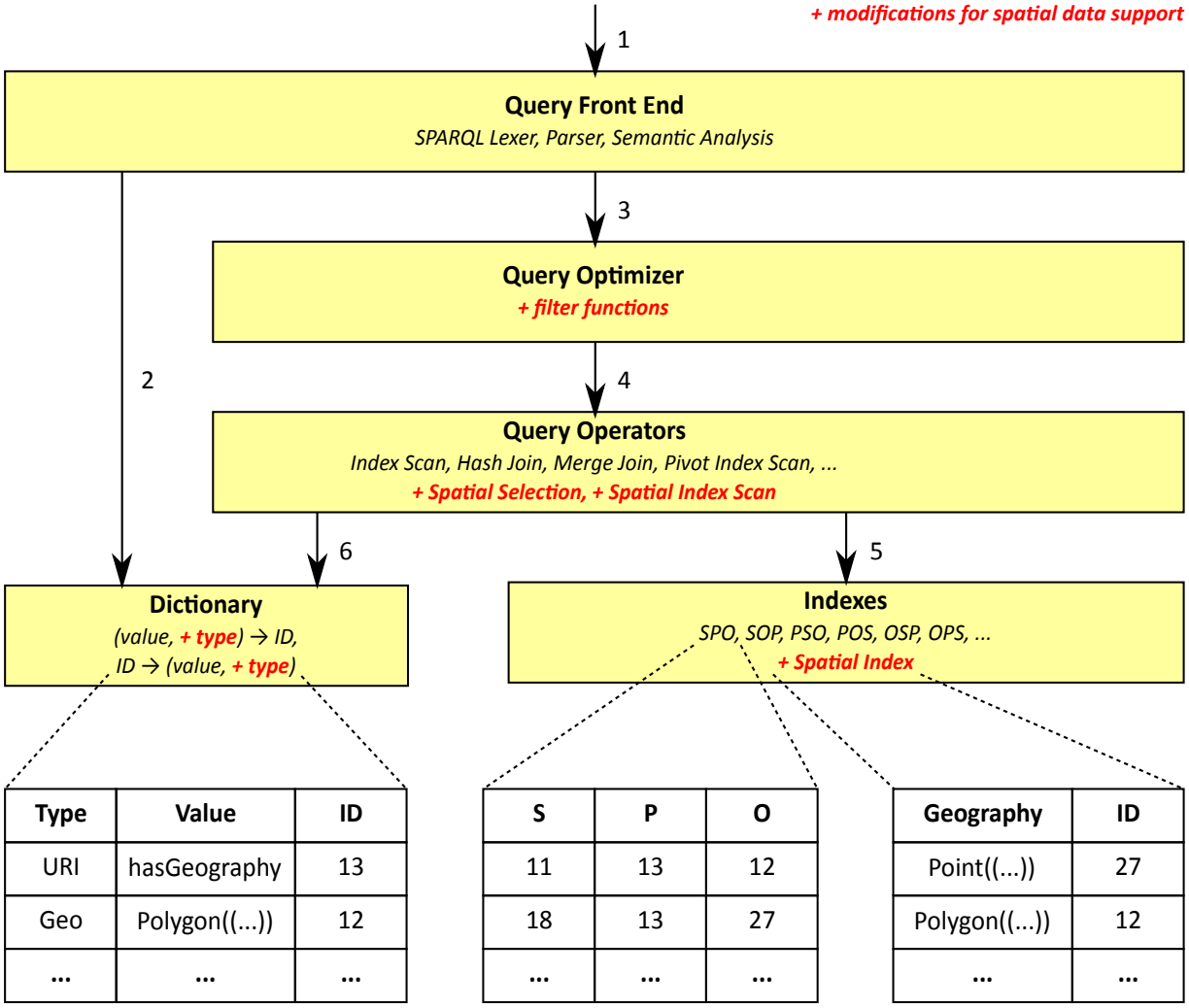


Figure 4.1: The architecture of the triple store (RDF-3X) and our modifications for deeply integrated support for spatial queries (marked with +)

4.3.2 Spatial Selection Operator

As our first step towards a triple store supporting spatial queries, we implemented a spatial selection operator that filters the results of a pure RDF pattern matching query. The selection operator supports all comparisons specified in the OpenGIS Simple Features Specification [Herring, 2006]. To implement the selection we used the GEOS C++ library,[5] which GIS systems, such as PostGIS, use as well. Rather than implementing the selection strictly on top of the triple store, we integrated the selection as an additional query operator. For this, we had to modify the query front end, as it neither recognized filter functions nor typed literals; RDF-3X does

[5] `http://trac.osgeo.org/geos/`

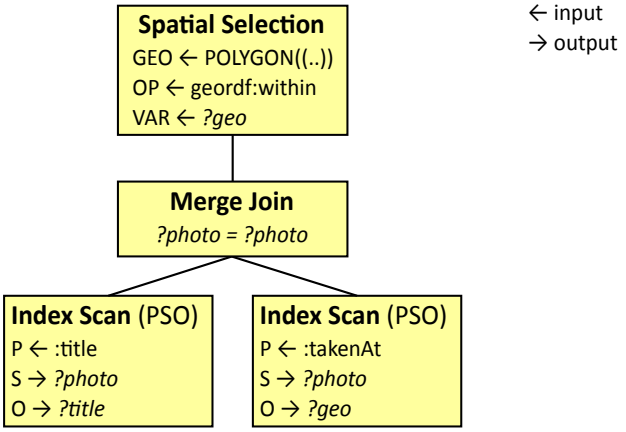


Figure 4.2: Execution plan for the query of Listing 4.7 using a spatial selection on top

not implement these parts of the SPARQL specification. We also had to modify the dictionary to store type information with literals. We did *not* modify the query optimizer. Instead, we simply put the spatial selection operator in front of the execution plan which the optimizer returns. Finally, we had to modify the result printer to display typed literals correctly with their type URI. Figure 4.2 illustrates an execution plan for the query of Listing 4.7 that uses the spatial selection.

RDF-3X does support SPARQL filters as long as they are restricted to identity comparisons (= and `!=`). This is because the integer IDs used in internal processing allow no further comparisons of the values. The spatial selection operator needs to perform non-trivial calculations to compare spatial features. There is no way but to look up every single ID from the dictionary and to restore the actual coordinates before the spatial predicate can be evaluated. As a dictionary look-up is a costly operation, we did not consider pushing the spatial selection down in the query graph. Performing the selection in the very end prevents unnecessary look-ups.

4.3.3 Spatial Index

The spatial selection on top of the pattern matching query enables the triple store to support spatial query predicates fully and is always applicable. Yet, it performs well only on queries with a restrictive graph pattern, which return a rather small number of features to compare. To select only those spatial features that are of interest to the query, a spatial index is needed (even if not every spatial index is applicable on all comparisons, e. g., disjoint). The spatial index maps the features to their dictionary IDs, so that other query operators can process them further. We implemented a spatial index based on the R-Tree index [Guttman, 1984] of the libspatialindex C++

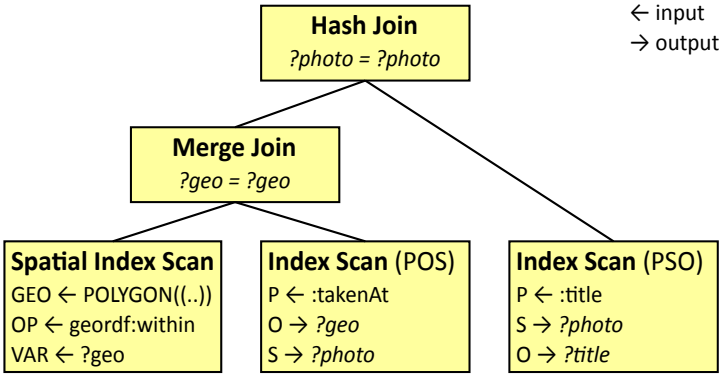


Figure 4.3: Execution plan for the query of Listing 4.7 using a spatial index scan

library.[6] The spatial index required deep modifications of the triple store. First of all, we had to modify the data import to recognize spatial literals and to insert them into the spatial index. From our work on the spatial selection, the query front end was already prepared to interpret the spatial filter functions. The essential (and most complicated) modification was to make the query optimizer generate plans which use the spatial index and join the results with other parts of the query.

The fastest join operation of the triple store is the merge join, which requires both operands to be sorted. However, the spatial index partitions the features based on geographic proximity and thus cannot return the feature IDs in a defined order. For this reason, our spatial index scan buffers all feature IDs in memory and sorts them before returning them. Subsequently, a merge join can intersect the feature IDs efficiently with RDF triples that are sorted by the variable which binds the features. The alternative to this sort-merge join would have been a hash join, which buffers all results in a hash table before joining them.

Figure 4.3 shows an execution plan for the query of Listing 4.7 which uses a spatial index scan. The spatial index scan returns exactly those IDs which correspond to the features matching the spatial query predicate; i. e. the features located within the polygon of the `FILTER` clause. The features are bound to the variable `?geo`, which occurs in the triple pattern `?photo :takenAt ?geo`. To determine valid bindings for the `?geo` variable, the features need to be joined with RDF triples containing the `:takenAt` predicate. An index scan on one of the six RDF triple indexes selects these triples. The query optimizer chooses the POS index, so that the predicate in question is selected and the resulting (object, subject)-pairs are sorted by `?geo`. Subsequently, a merge join combines these pairs with the feature IDs, which were sorted by `?geo`, too. The resulting set of photo URIs (`?photo`) and spatial features (`?geo`), is still sorted by `?geo` and must be joined with the photo titles. The titles cannot be sorted

[6] `http://sourceforge.net/projects/spatialindexlib`

by `?geo`, which rules out a merge join. Instead, a hash join completes the query result. (If further attributes of the photos were retrieved, it might be worth considering a nested loop join and a pivot index scan, as explained in Chapter 3).

4.3.4 Storing the Features

As described in Section 4.2.1, our approach expresses spatial features as literals of a complex spatial type. The features are formulated in the Well-know Text (WKT) format and need to be parsed to evaluate a query predicate on them. This is the case both for the spatial selection and for the spatial index. The index only selects candidates based on their bounding box; a second refine step is required to evaluate the query predicate exactly. In order to accelerate the parsing of the features, we store them in the dictionary and in the spatial index by means of the Well-know Binary (WKB) format [Herring, 2006]. WKB consumes significantly less space and is much more efficient to parse. The downside of this approach is that the features must be converted back to WKT before the final query result can be returned. Thus, whenever the dictionary resolves the ID of a spatial feature, it recognizes its type and parses the WKB string into a geography structure. Then it serializes the object to a WKT string and discards the geography structure.

4.4 Evaluation

We carried out extensive performance measurements. Our approach combines RDF triple store technology with spatial data processing to a new kind of system, which makes it difficult to compare to other systems. A relational spatial database is likely to perform better, as the database schema may model resources as complete records. This avoids most of the joins which a triple store requires as the price to pay for the schema flexibility (and other advantages) of RDF. Comparisons to other triple stores are not applicable, as they lack arbitrary spatial functionality. Moreover, our implementation builds on RDF-3X, which has been extensively evaluated in literature [Neumann and Weikum, 2010; Atre et al., 2010]. Instead, we focused on illustrating the specific characteristics of our implementation and evaluated the effect of our modifications to RDF-3X.

4.4.1 Test Setup

The test data we used in our evaluation follows the schema of OpenStreetMap (OSM), as shown in Figure 4.4. In OSM, every spatial resource is a Node. A Node carries a geographic location and a number of Tags. The Tags describe the Node through

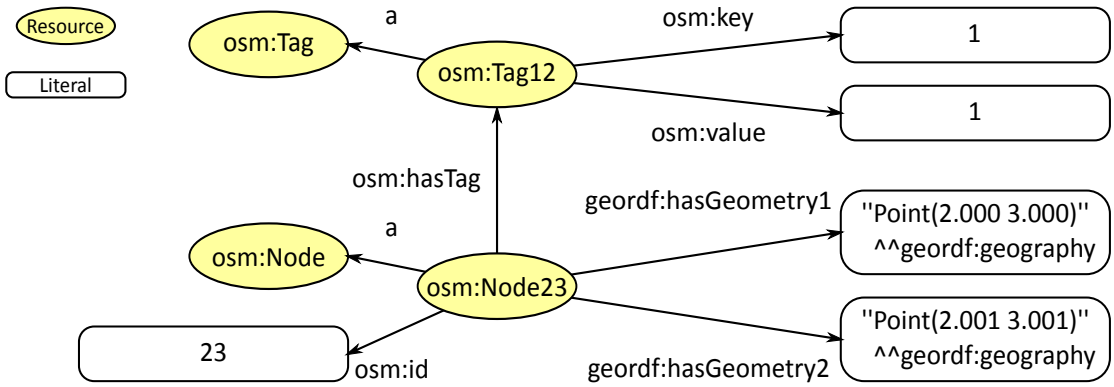


Figure 4.4: The data model used in our evaluation follows the schema of OpenStreetMap (OSM)

their key and value attributes, which are arbitrary strings. We imported data from OSM and converted it to RDF with spatially typed literals. Moreover, we generated large amounts of artificial test data, which makes it easier to estimate the number of features which match a spatial query predicate. For this we created Node resources which are located on a grid. The Nodes carry one or more spatial features, which are points on the grid. The Nodes own the features exclusively, i. e., there is a 1:1 or a 1:n ratio between Nodes and points. To evaluate an RDF pattern matching part in queries, we generated Tags which simply carry integer values. To achieve different selectivities of RDF patterns, we tagged *every* Node with key 1, every *second* Node with key 2, every *fourth* Node with 4, etc., up to 1024. This enables us to select a fraction of $\frac{1}{2^{key}}$ ($key \in \{1..10\}$) of all Nodes. We used two basic grid sizes. The *small grid* contained 1.05 million Nodes which resulted in 11.5 million RDF triples and database files of 856 MB size. The *large grid* contained 104.88 million Nodes, 1.15 billion RDF triples and made up database files of 89.29 GB.

We measured the end-to-end execution time counted from the time of query submission to the time including outputting the final results (except for the dictionary test in Section 4.4.3). We ran all queries on cold and warm caches. For cold caches we dropped the file system caches of the operating system before each query. For warm caches we ran the query once before measuring the time. We measured all queries ten times and report, on a logarithmic scale, the median of ten test runs.

We ran the tests on a desktop PC and on a mobile device. In both cases the test machines were the same as the ones described in more detail in Section 3.5.2: A Dell Optiplex 755 (Intel Core2 Quad Q9300 CPU at 2.50 GHz and 4 GB of main memory) and a *Nokia N900* smartphone (ARM Cortex-A8 processor at 600 MHz and 256 MB of Mobile DDR main memory). On the mobile device we only used the test databases containing the *small grid*.

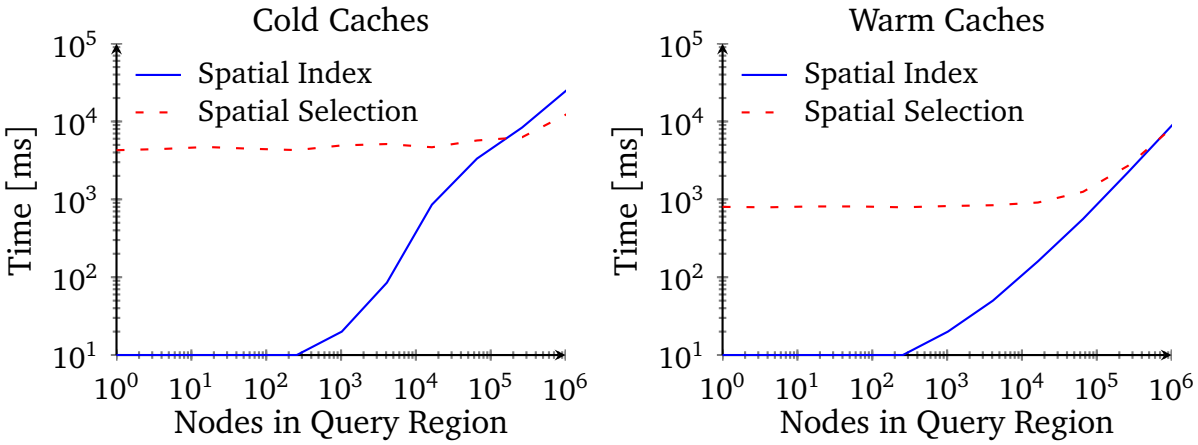


Figure 4.5: Spatial Selection vs. Spatial Index on the **desktop PC**

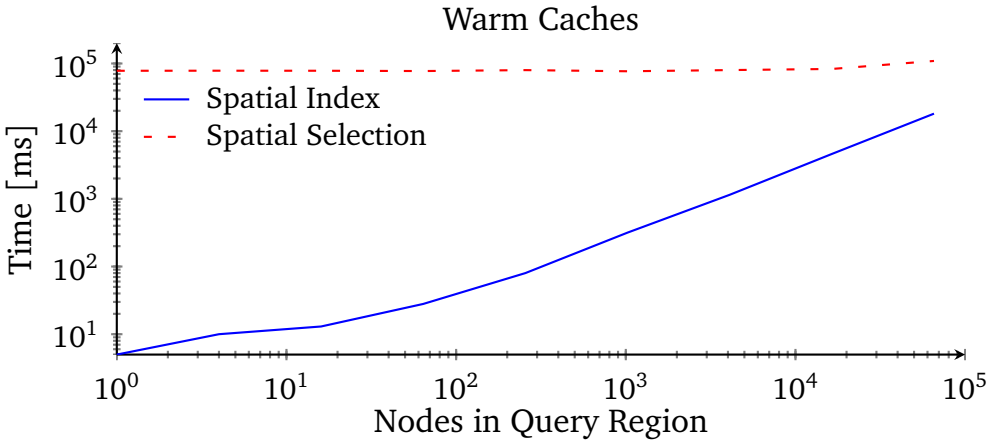


Figure 4.6: Spatial Selection vs. Spatial Index on the **mobile device**

4.4.2 Spatial Selection vs. Spatial Index

First, we compared the spatial selection operator with the spatial index. As discussed in Section 4.3, the selection must evaluate the spatial predicate on all features returned by the pattern matching query. The index selects only the relevant features for further processing. We ran the query of Listing 4.8 with different query regions (polygons) on the small grid of Nodes. The RDF pattern returns all Nodes in the database, so the query is only selective on the spatial filter.

Figure 4.5 and Figure 4.6 show the results on the desktop PC and on the mobile device, respectively. It is obvious that the spatial selection performs equally for all query regions, as the pattern matching part always returned all stored features to the selection. For small regions, fewer query results are produced, but the selection

```
SELECT * WHERE {
    ?node  geordf:hasGeography  ?geo.
     FILTER geordf:within (?geo,
        "POLYGON(( «Query Region» ))"^^geordf:geography)
}
```

Listing 4.8: Query to compare the spatial selection to the spatial index

already resolved the respective IDs from the dictionary, so that they are always cached. On the desktop PC the spatial index is faster for up to about half a million selected Nodes, especially on cold caches. Beyond that point, the costs for joining the intermediate query results dominate the selection. On the mobile device the picture is similar, except that the selection is a lot slower than the spatial index across the entire measured range. The hardware of the mobile device is simply not designed for large volume data processing, so that evaluating all 1.05 million Nodes in the spatial selection is never faster than finding up to 10^5 Nodes from the spatial index. Note that this strongly depends on how many tuples the graph patterns produce. A real-life query, on the other hand, is likely selective on the RDF pattern too, so that the break-even-point of the spatial selection and the spatial index may be a lot lower.

4.4.3 Dictionary Performance

As outlined in Section 4.3.4, we modified the dictionary to store spatial features in the Well-know Binary (WKB) format for faster parsing. The downside is that the geometries must be parsed in all cases, i. e., also to print them as typed literals in the Well-know Text (WKT) format. We ran a series of tests to determine the parsing overhead. Using the query of Listing 4.9 we produced different amounts of IDs for the dictionary to resolve to URIs or spatial literals. We used the generated large grid database and a real-world data set from OSM of similar size. OSM contains more complex features, such as polygons and linestrings, in addition to points. In contrast to all other tests, we did *not* record total execution times in this test, but measured only the time to resolve the IDs. Figure 4.7 and Figure 4.8 report the median of ten runs on the desktop PC and on the mobile device, respectively.

```
SELECT «?node | ?geo» WHERE {
    ?node  geordf:hasGeography  ?geo.
} LIMIT «Number of Dictionary Entries»
```

Listing 4.9: Query to measure the dictionary performance

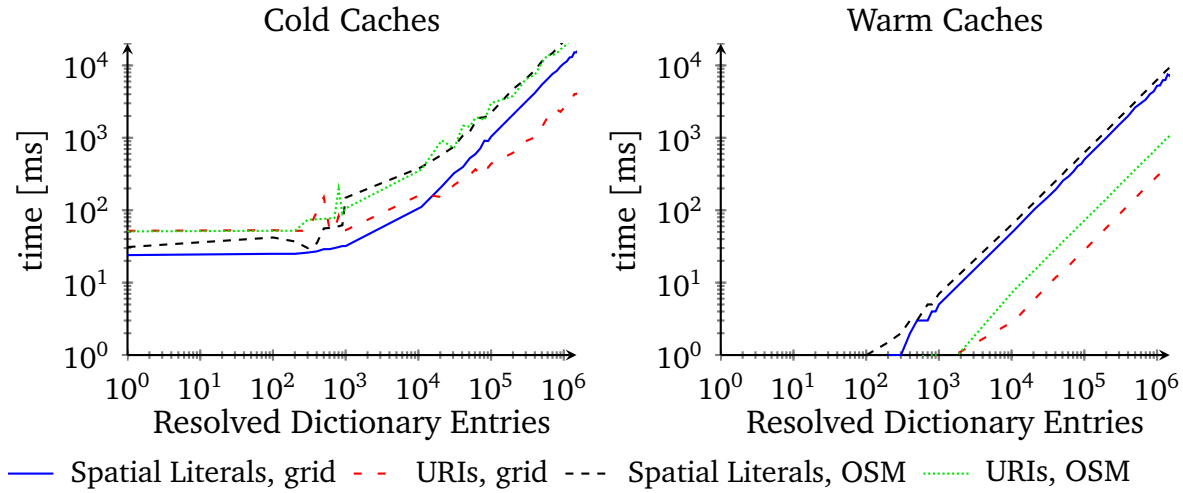


Figure 4.7: Dictionary performance: large grid and OpenStreetMap (OSM) data on the **desktop PC**

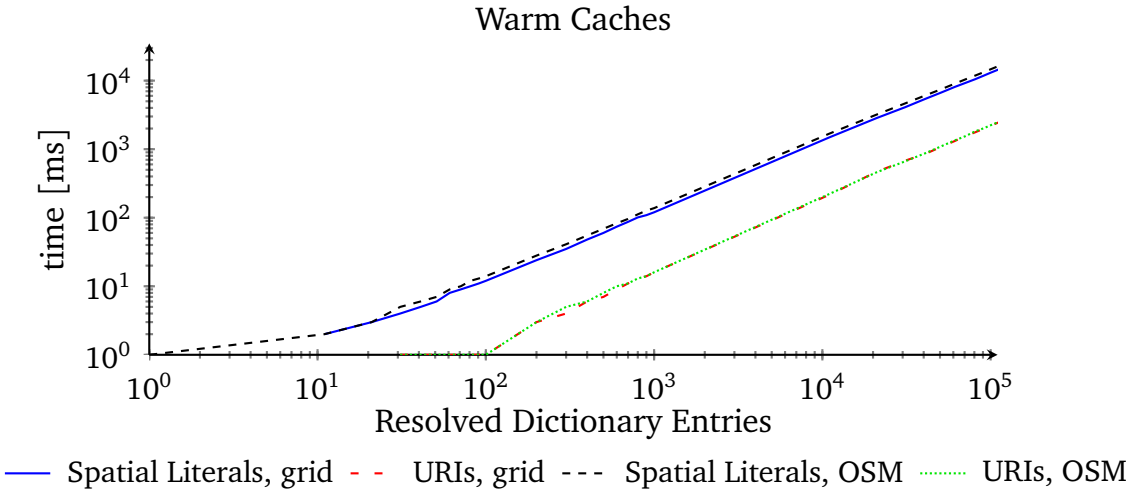


Figure 4.8: Dictionary performance: small grid and OpenStreetMap (OSM) data on the **mobile device**

The dictionary requires exactly two page reads to look up any ID. Moreover, the IDs are resolved in ascending order, causing an ideal cache hit ratio. Thus, for small amounts of IDs the lookup time is very small on cold caches and hardly measurable on warm caches on the desktop PC. The overhead to parse spatial literals is observable, but only starts to play a role for very large ID sets. For cold caches, the characteristics of the data, e. g., page locality, dominate parsing: the grid URIs perform better than OSM URIs. Also, the more complex OSM literals take negligibly longer to parse. On

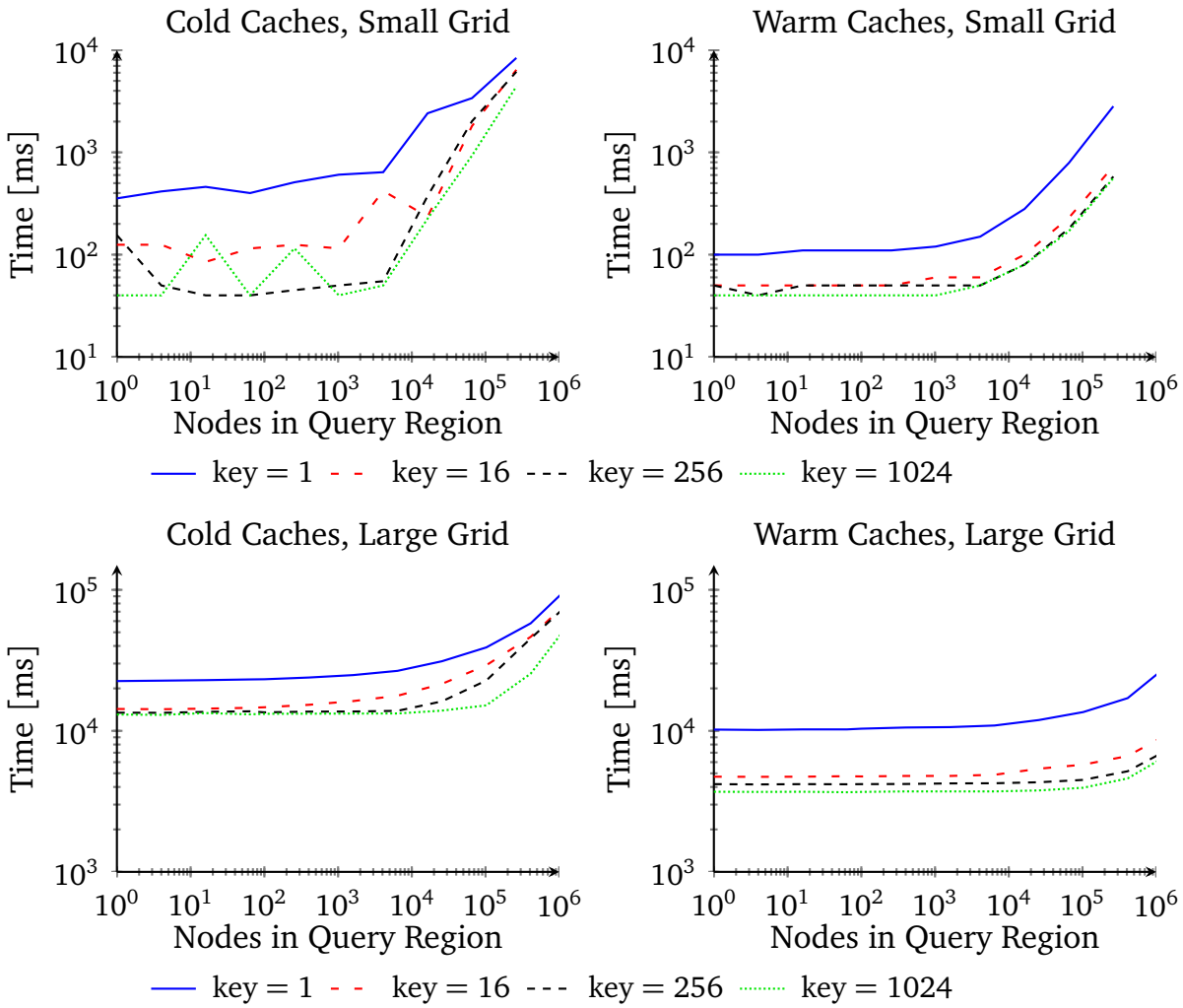


Figure 4.9: Different selectivities in the RDF pattern matching part and in the spatial predicate of a query on the **desktop PC**

the mobile device the dictionary lookup times are always measurable, yet still very fast. The parsing overhead of the spatial literals is observable as on the desktop PC.

Atre et al. [2010] observed the performance impact of resolving very large query results. Our measurements confirm this. The impact of parsing WKB literals in addition to resolving the literals is only a tiny fraction of the overall execution time and included in all reported times in the evaluation of this chapter.

4.4.4 Different Selectivities

We ran a series of queries with different selectivities in the RDF pattern matching part and in the spatial query predicate. We queried Nodes with a particular Tag

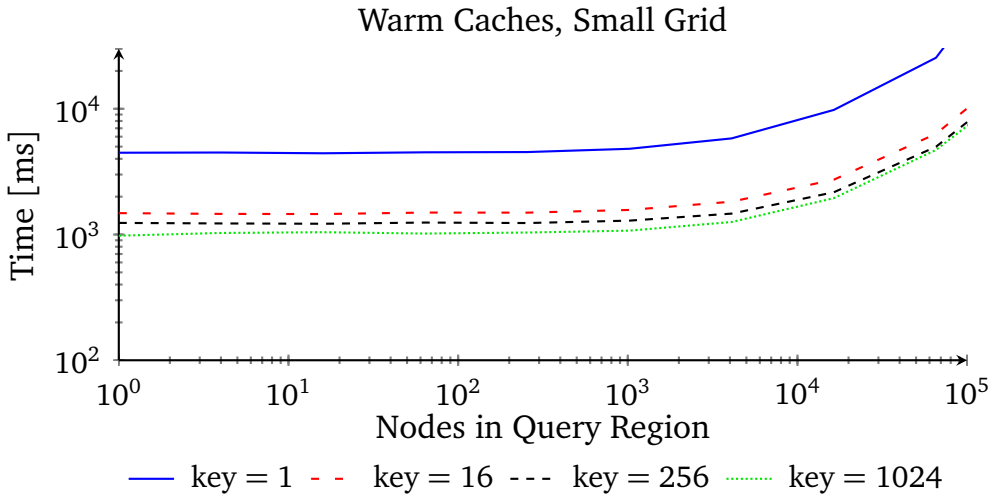


Figure 4.10: Different selectivities in the RDF pattern matching part and in the spatial predicate of a query on the **mobile device**

key to influence the pattern matching part, as every Tag key occurs at a different fraction of the Nodes (see Section 4.4.1). We influenced the spatial selectivity through different query regions and evaluated the spatial query predicate on the spatial index. Listing 4.10 shows the test query. Note that the query resolves and prints all bound variables. Figure 4.9 shows the results on the desktop PC, both for the small and the large grid database. Figure 4.10 shows the execution times for the small grid as measured on the mobile device.

Much more than selectivity does the database size make an impact. Even though RDF-3X is good at discarding unnecessary intermediate tuples early [Neumann and Weikum, 2010; Atre et al., 2010], the larger database inevitably causes more of them. For up to 10^4 resp. 10^5 Nodes, the results are nearly independent of the query region on the desktop PC. On the mobile device the figures are very similar, except that they are about ten times slower. The curves indicate that the query plan did not use the spatial index optimally. Joining the spatial index with the rest of the query

```
SELECT * WHERE {
    ?tag   osm:hasKey            " «Tag Key» ".
    ?node  osm:hasTag            ?tag.
    ?node  geordf:hasGeography   ?geo.
     FILTER geordf:within (?geo,
        "POLYGON(( «Query Region» ))"^^geordf:geography)
}
```

Listing 4.10: Query to compare performance with different selectivities

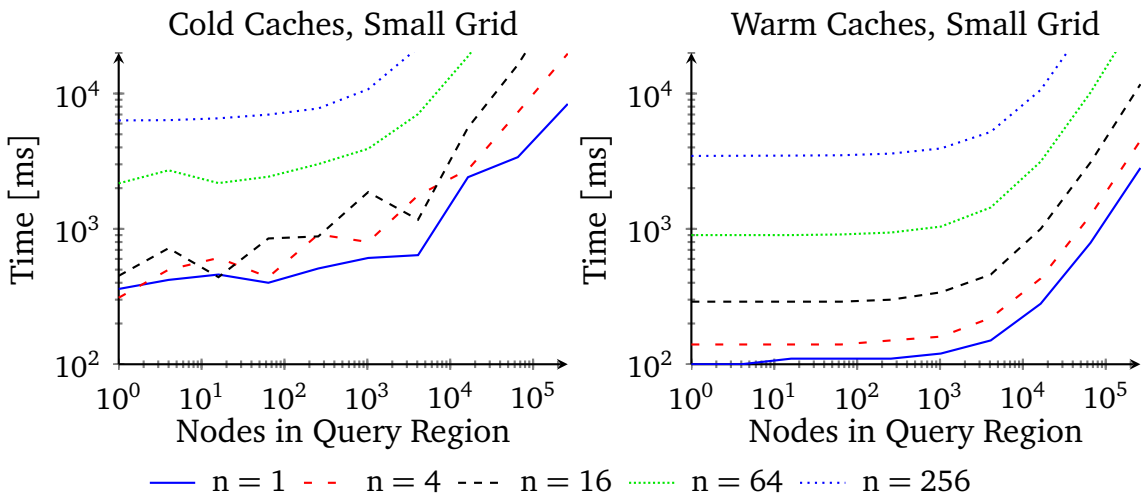


Figure 4.11: Multiple spatial features per resource: Every Node possesses *n* different features on the **desktop PC**

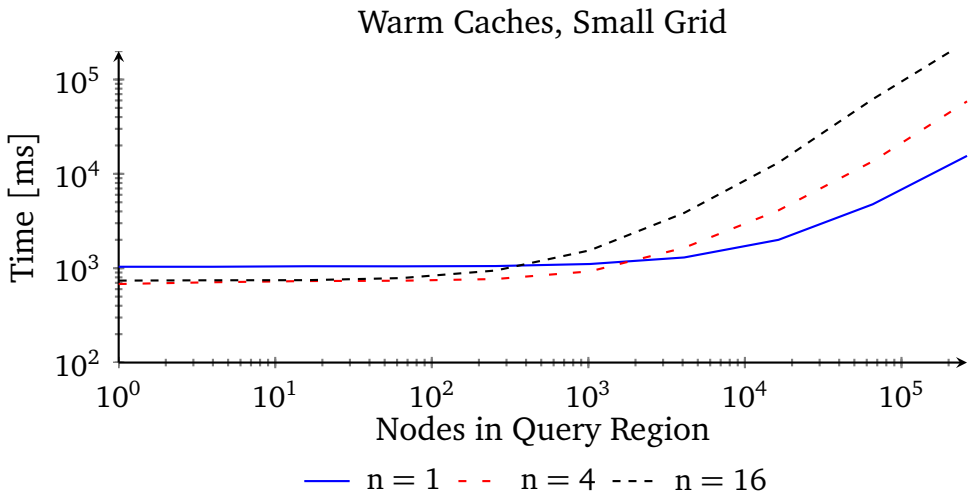


Figure 4.12: Multiple spatial features per resource: Every Node possesses *n* different features on the **mobile device**

earlier would avoid intermediate tuples. Better cardinality estimation would help the query optimizer create better execution plans in this case, which we address in Section 3.3.4. In addition to that, it is observable that the less selective Tag keys create higher load. However, the overhead is much smaller than the difference in selectivity. Generally, we think the shown performance is very good.

```
SELECT * WHERE {
    ?tag   osm:hasKey            "1".
    ?node  osm:hasTag            ?tag.
    ?node  geordf:hasGeography1  ?geo.
     FILTER geordf:within (?geo,
        "POLYGON(( «Query Region» ))"^^geordf:geography)
}
```

Listing 4.11: Test query to select one out of multiple features per resource

4.4.5 Multiple Spatial Features per Resource

In a final series of tests we address the situation of multiple spatial features per resource. It depends on the data model whether this can occur. Models such as OSM or those using the place resources discussed in Section 4.2.1 contain only 1:1 relationships between resources and features, i. e., every feature belongs to exactly one resource. Other models may contain 1:n relationships, e. g., to model both the center point and the boundary of a building. Our spatial index contains all features of the entire RDF graph in a single index structure. Thus, it returns all features that match a spatial query predicate, regardless of the resource it belongs to or the predicate connecting it to the resource. I. e. both the center point and the boundary of the aforementioned building are returned if they match the spatial predicate—even if the query only needs the center point.

To measure this effect, we generated different variants of the *small* grid database with a different number n of points per Node, resulting in database sizes of up to 33.9 GB. In addition to that, all points of the same resource used different predicates (`osm:hasGeography1`, `osm:hasGeography2`, ...). Listing 4.11 shows the corresponding test query; to include further pattern matching parts, it selects Nodes carrying Tag 1. Figure 4.11 and Figure 4.12 show the results. Note that the X axis counts Nodes, not features. Especially for cold caches, a higher amount of features per resource clearly does increase the costs in our indexing approach. However, even though more features per resource take longer, the effect is moderate for queries which select a small to medium amount of features, both on the desktop PC and on the mobile device.

4.5 Cardinality Estimation

The query optimizer of an RDF triple store with support for spatial query processing is confronted with a considerable number of decisions when creating an execution plan

for a spatial query. The query optimizer must decide whether a spatial query predicate should be evaluated using the spatial selection or the spatial index. Moreover, choosing a good join order is utterly important, also for joining the spatial index. In order to make sensible decisions, the query optimizer needs to estimate the costs of (parts of) different query plans. To estimate these costs the query optimizer needs an estimate on the cardinality of the query plans.

Cardinality estimation is a general challenge in RDF triple stores, as discussed in Chapter 3. Triple stores cannot assume schema information but must work at triple granularity, even if the triples actually describe complex resources. This leads to statistical dependencies between the triples which complicate cardinality estimation considerably. E. g. if a resource carries the attribute `firstName`, it is far more likely that it possesses a `lastName` than a `startDate`; and relations such as `employedBy` or `marriedWith` likely return far less join results than `takesPart` or `knows`.

If the triple store supports spatial query processing, it must additionally consider spatial distributions and dependencies. The distribution of spatial features managed by the triple store is presumably very uneven. Large parts of the earth's surface are oceans, deserts, or forests and the distribution of spatial features in these areas is likely less dense than in metropolitan areas. This distribution can be used for cardinality estimation. However, this assumes that the spatial features which a query selects are distributed the same way as all spatial features in the database. E. g. the home address of a friend is more likely to be located in a populated area than in the middle of the Atlantic ocean. The populated area is likely to contain generally a lot more spatial features than the ocean. However, the home address of a friend also likely depends on where the user lives—and there will be many densely populated areas in the world where the user knows nobody. I. e., in this case the distribution of the friends is different than the general spatial distribution of the database. Thus, in addition to the spatial distribution, cardinality estimation also requires considering dependencies with non-spatial information. In summary, cardinality estimation in RDF triple stores supporting spatial queries must address three basic challenges:

1. Cardinality estimation for spatial features in a given area
2. Cardinality estimation for arbitrary RDF graph patterns
3. Cardinality estimation for spatial features in a given area that are connected to a given RDF graph pattern.

The first two challenges have been addressed in prior research, whereas the third challenge has not yet been considered in detail. In this section we present an approach to estimate the cardinality of RDF graph patterns that are connected to a spatial

feature in a given query area. As opposed to the other sections in this chapter, we restrict our focus to containment queries, i. e. queries that search for spatial features in a given area. This is the most relevant use case and the basis for more complex spatial query predicates, such as *touches*, *crosses*, etc. We give an overview on prior research (Section 4.5.1), introduce our approach (Section 4.5.2), and evaluate it using real-world data from OpenStreetMap (Section 4.5.3).

4.5.1 Related Work

Cardinality Estimation for Spatial Features in a Given Area

The predominant methods for estimating cardinality and selectivity of spatial queries are based on two-dimensional histograms. These methods can be divided into 1) data grouping and 2) cell density.

Data Grouping Data grouping techniques group similar spatial features into buckets and assume that all features in the same bucket are distributed uniformly (uniformity assumption). Equi-partitionings group features into buckets that are equal according to some measure, e. g. equal numbers of spatial features or equal areas. Index-based grouping techniques exploit the partitionings created by a spatial index, such as the R-tree [Guttman, 1984], which attempts to minimize the area, margin and/or overlap of its internal nodes. Skew-aware groupings, such as Min-Skew [Acharya et al., 1999], create binary space partitionings such that the negative effects of the uniformity assumption in presence of spatial skew are minimized.

Data partition techniques may or may not allow the geographic regions of the buckets to overlap. As the number of non-overlapping equal size buckets grows exponentially with increasing dimensions in the data, Gunopulos et al. [2005] propose an approach named *GENHIST* (for GENeralized HISTograms). GENHIST uses overlapping variable-size buckets to address skew in high-dimensional data. Dense areas are covered by several buckets and their data density is approximated as the sum of the data densities of the individual buckets. This way the uniformity assumption can be applied without requiring a large number of small buckets.

Note that the buckets do not necessarily cover the entire space. This deals well with empty areas. On the other hand it is not as straight forward to find the relevant buckets for a query if they are not of equal size or not equally distributed.

Cell Density Techniques based on cell density divide the space into a number of disjoint, adjacent, and equal-sized grid cells. Thus, unlike data grouping techniques, they cover the entire space. Different approaches have been published on what

information is recorded about each cell. Jin et al. [2000] proposed the Cumulative Density (CD) approach, which allows estimating cardinality of queries on axis-aligned rectangles. For each cell (i, j) it records how many rectangles have their lower left, upper left, lower right, and upper right corner in the area between (0, 0) and (i, j), respectively. Using these values the cardinality of any containment query can be estimated in constant time, regardless of the query area or the cell size. Most notably it is not required to examine all cells that overlap with the query region.

Beigel and Tanin [1998], Sun et al. [2006] and Lin et al. [2003] proposed cell density approaches based on the Euler Formula [Harary, 1969], allowing the topological relations *contains*, *contained*, *overlap*, and *disjoint*.

PostGIS[7] uses a comparatively simple approach based on cell density. First, it determines the bounding box of the entire database. This bounding box can be shrunk by a given factor, assuming that the majority of features is not located in peripheral areas. The bounding box is divided into grid cells and for every cell the number of overlapping spatial features is determined. Thus, features other than points (line strings, polygons, etc.) are counted once in every cell they overlap with. Last, the number $\bar{c}$ of cells is computed with which a spatial feature overlaps on average (if only points are stored, then $\bar{c} = 1$). To estimate the cardinality of a containment query, the feature counts of all cells overlapping with the query region are summed up. Partially overlapping cells are considered according to the fraction by which they overlap. Finally the resulting sum is divided by $\bar{c}$ to cope with features that were counted multiply. The advantage of this approach lies in its simplicity, which caters for a very fast implementation. The downside is that it does not well address very skewed data. In our experiments on world-wide data [Dick, 2011, page 90] the majority of cells was empty, because they were located in the ocean or desert.

Cardinality estimation for arbitrary RDF graph patterns

Triple Join Statistics To address the statistical dependencies between individual RDF triples, Neumann and Weikum [2009] approximate the join selectivity of two triple patterns. Their approach is based on the observation that joining the triples matching the triple patterns t_1 and t_2, respectively, is equivalent to joining t_1 with the entire database $\mathcal{T}$ followed by a filter on t_2.

$$t_1 \bowtie t_2 \equiv \sigma_{t_2}(t_1 \bowtie \mathcal{T})$$

The latter form would be very inefficient for query execution. However, the selectivity of the selection σ_{t_2} can be computed easily from the result cardinality of t_2, which is

[7] `http://postgis.refractions.net/`

available in the aggregated indexes, as described in Section 3.3.5. The cardinality of the join $t_1 \bowtie \mathcal{T}$ is precomputed using a summation on the aggregated indexes. The join statistics are exact if the independence assumption between the triple patterns holds. When the triple patterns are not independent, this creates an estimation error which is not as bad as multiplying the individual selectivities of t_1 and t_2. Nevertheless, as the join statistics are applied every time the bottom-up query optimizer creates a join, as described in Section 3.3.1, the estimation error for the entire execution plan may grow up to the point where the estimates are useless.

An important lesson to learn from this is that the cardinality of RDF graph patterns must not be estimated at triple level, as repeatedly applying such statistics renders them unusable when creating query plans incrementally. Markl et al. [2005] call this behaviour "fleeing to ignorance". Instead, larger structures must be addressed, which the query optimizer may ideally apply as a whole.

Graph Summarization Maduko et al. [2008] propose an approach to summarize graph patterns in arbitrary graphs (i. e. not necessarily RDF). They compute and store the frequency of subgraphs up to a given size. If the synopsis is too large, they prune the path length such that the error resulting from the independence assumption is minimized. They use the known fragments for cardinality estimation and combine them assuming statistical independence. However, Neumann and Moerkotte [2011] observed that for real-world RDF data, the chaotically distributed triple predicates lead to a combinatoric explosion. Thus, even for small path lengths, the number of subgraphs grows too large and the pruning techniques cause severe estimation errors.

Characteristic Sets As introduced in Section 3.3.4, Neumann and Moerkotte [2011] capture statistical dependencies between RDF predicates by reconstructing the entities which the RDF triples represent. Up to a given maximum, Neumann and Moerkotte simply enumerate all sets of predicates which occurred together at a subject. Thus, they index all predicate sets of star-shaped structures in the entire RDF graph. This is based on the observation, that in practice SPARQL queries tend to have variables in the subject or in the object part of a triple pattern, but infrequently in the predicate. To generate an execution plan for a given SPARQL query, they search for star-shaped structures in the query and determine all Characteristic Sets which include them. The occurrence counts of these Characteristic Sets are added up and thus return the number of resources in the database that match the star-shaped structure of the query. The output cardinality of the query part corresponding to the star structure is the number of matching resources times the factor by which cross-products over multi-attributes increase it. This factor is estimated by averaging

multi-attribute occurrences across all Characteristic Sets containing the star structure, i. e. attribute values are assumed to be distributed equally. To estimate joins *between* star-shaped structures, Neumann and Moerkotte use the triple join statistics of Neumann and Weikum [2009] where possible, and assume independence otherwise.

Discussion

Histogram-based cardinality estimation techniques for spatial queries all have in common that they create axis-aligned rectangular buckets or cells. Approaches differ in the kinds of information they store for each bucket, whether buckets are equal-sized, whether they may overlap, or whether they cover the entire space. They are usually not restricted to two dimensions, albeit not optimized for high-dimensional data. One approach to estimate the spatial features in a given area that are connected to a graph pattern might be to model the adjacent RDF subgraphs as further dimensions of the spatial feature. The problem with this is the heterogeneous and string-oriented nature of RDF. The number of dimensions is unknown and much higher than the numbers typically applied on multi-dimensional histograms (Gunopulos et al. [2005] evaluated GENHIST for up to 10 dimensions).

The lesson to learn from the literature on cardinality estimation for RDF queries is that the statistical dependencies originating from the decomposed structure of RDF cannot be addressed at triple level. Instead, larger structures must be built and estimated, such as the subgraphs of Maduko et al. [2008] or Characteristic Sets [Neumann and Moerkotte, 2011]. At this, one must avoid the combinatoric explosion from the number of subgraphs. Another lesson learned from Characteristic Sets is that it is sufficient to consider the predicates of the RDF triples.

Our approach, which we present in the following section, uses separate techniques for spatial histograms and RDF cardinality estimation. First, the spatial features are grouped into buckets. Then, the adjacent RDF subgraphs are summarized for each bucket using a synopsis which we call *path bundles*.

4.5.2 Approach: Buckets and Frequent Path Bundles

Our approach to estimate the number of spatial features in a given area that are connected to a given RDF graph pattern is based on the following three principles:

1. Dividing the geographic space into two-dimensional histogram buckets.

2. For each bucket, approximating the most frequent subgraphs that are connected to a spatial feature in that bucket.

3. Sum up the cardinality estimation of all buckets covered by the query region; considering partially overlapping buckets to the fraction by which they overlap.

To create the histogram buckets, any approach to divide the space or to group spatial features into buckets can be used. In our evaluation we use the cell-density approach of PostGIS explained above, despite its shortcomings for the sake of simplicity. Summarizing the most frequent subgraphs adjacent to a bucket is not as straight forward. It is crucial to capture larger structures to address statistical dependencies in the data while avoiding the combinatoric explosion at the same time.

Preliminaries

Note that we do not address cardinality estimation of arbitrary RDF graph patterns anywhere in the RDF graph. Instead, the spatial features in a particular bucket give us a set of well-defined entry points. Moreover, we assume that the correlation of RDF subgraphs with the bucket (i. e. the spatial area) decreases with the distance in the graph. Intuitively, the number of photos taken in a particular area, for instance, depends a lot on the area. But structure of the information that is stored *about* a photo, hardly differs between areas. Thus, starting from a spatial feature f in the bucket b, we only consider the subgraph adjacent to f up to a maximal search depth d. Beyond that, we rely on general RDF cardinality estimation techniques.

To approximate the subgraphs adjacent to b, we could simply record the frequency of the Characteristic Sets that link to any f in b. This would cover the subgraphs up to depth $d = 2$. This has, however, two disadvantages. First, it ignores the RDF predicate via which a particular resource is connected to f, e. g. *locatedAt*, *livesAt*, or *worksAt*. This predicate is the part of the subgraph that is closest to f and, according to our assumption above, the most important part. Second, the data might be modeled using *place resources*, as described in Section 4.2.1. In this case the Characteristic Sets would only cover the place resources, which are fairly uniform. The correlation between f and the subgraph, which is the important part, would be estimated using the independence assumption. Even worse, the data set may have been integrated from different sources which partly use place resource and partly don't (after all, the flexibility of RDF for data integration is one of its key selling points). For such a scenario, more flexibility towards the d would be desirable.

Path Bundles

To estimate the cardinality of subgraphs adjacent to a spatial histogram bucket b we consider subgraphs which

1. start with a spatial feature f that belongs to b
2. are trees, with f as their root
3. have maximal tree height d.

To approximate which of these subgraphs are matched by a query, we bundle together those paths which also occur in the query and estimate cardinality on the path bundle. We must take into account the combinatoric explosion of subgraphs and paths, the direction of triples, and statistical dependencies between paths.

Fighting the Combinatoric Explosion As the number of such subgraphs grows exponentially with d, we do not store their exact structure. First of all, we only record the predicates of the corresponding RDF triples and ignore resource URIs and literal values. This helps to collapse many isomorphic subgraphs into one representation, just like with Characteristic Sets. Second, we only record the cycle-free paths from f through the subgraphs, which effectively creates a tree. Presumably, many non-isomorphic subgraphs share common paths, especially as we do not only record paths with maximal length d, but also all their "prefixes". Thus, by recording only paths instead of entire subgraph structures, we can further reduce the data volume.

Considering the Direction of Triples When determining the tree of paths that are reachable from f we should take into account the direction of the RDF triples. f is always the object of a triple $t_0 = (r_1, p_0, f)$. I. e., the first step of any path is always in the opposite direction of t_0. We record the predicate p_0 in this step and follow the second step of the path tree from resource r_1. The second step incorporates all triples that are adjacent to r_1 (except for t_0). Now these triples may either originate *from* r_1 or lead *to* r_1. I. e., in the second step we follow all outoing triples $\overrightarrow{t_1} = (r_1, \overrightarrow{p_1}, \overrightarrow{r_2})$ as well as all incoming triples $\overleftarrow{t_1} = (\overleftarrow{r_2}, \overleftarrow{p_1}, r_1)$. Thus, when recording the predicates of a path we do not only record the predicates $\overrightarrow{p_1}$ and $\overleftarrow{p_1}$, but also memorize the direction in which they were followed from f. I. e., we really store the pair $p_{dir} = (p, dir)$ of the predicate value p, e. g. *livesIn*, and the direction $dir \in \{\rightarrow, \leftarrow\}$. p_{dir} can be considered the edge label of an undirected graph and thus a new kind of predicate. As all URIs and literals are internally represented as integer IDs, we also create an integer for p_{dir} using a simple transformation. We apply the same transformation on the predicates of a query when looking for paths to estimate.

In many cases a predicate possesses a natural direction. A person, for instance, always lives in an apartment and not vice versa. Also, we only consider paths that start at a spatial feature, so that we encounter many triples only in one direction.

$$\begin{array}{llllllll} \pi_1 & = & f & \underline{locatedAt} & a & \underline{livesIn} & p & \underline{firstName} & n \\ \pi_2 & = & f & \underline{locatedAt} & a & \underline{livesIn} & p & \underline{eMail} & e \end{array}$$

Listing 4.12: Sample paths starting at a spatial feature f (triple predicates underlined)

Thus, by recording p_{dir} instead of p we do not face the exponential growth of paths adjacent to spatial features that is theoretically possible. Nevertheless, considering the direction of predicates we approximate the subgraph structure more precisely. This is, by the way, different to Characteristic Sets, which consider only outgoing triples. For simplicity of presentation, we will refer to p_{dir} as a normal triple predicate in the remainder of this chapter.

Addressing Statistical Dependencies The catch with decomposing subgraphs into paths is that paths are not statistically independent. Multiplying the selectivity of the paths π_1 and π_2 of Listing 4.12 will lead to a severe underestimation. In many cases they refer to the same person p living in the same apartment a, to two different persons who live in the same apartment, or even to two different apartments at the same location (e. g. in different floors of the same building). However, as we only consider the predicates of the RDF triples, we loose this information. Thus, we must not forget which paths occurred together, i. e. started at the same spatial feature f.

For this purpose, we store a placeholder for every spatial feature f together with every path that started at f. This results in a relation $B \subseteq P \times F$ for each bucket, with F as the set of all spatial features of the bucket and P as the set of all adjacent paths. B maps every path $\pi \in P$ to the set of features F_π at which this path occurred in the bucket. Moreover, $\left|F_\pi\right|$ gives us the occurrence count of π. To deal with statistical dependencies of any set of paths π_i, we can intersect the corresponding features F_{π_i} to determine those features, at which these paths occurred together. Thus, the relation B allows to combine the paths to a *path bundle* in order to emulate the original subgraph structures. Most notably, the cardinality of the intersected feature sets results in the correct number $card_f$ of features in the bucket that all carry the paths π_i. We implemented the sets F_{π_i} as bit vectors, so their intersection can be computed very efficiently.

$$card_f = \left| \bigcap_i F_{\pi_i} \right| \tag{4.1}$$

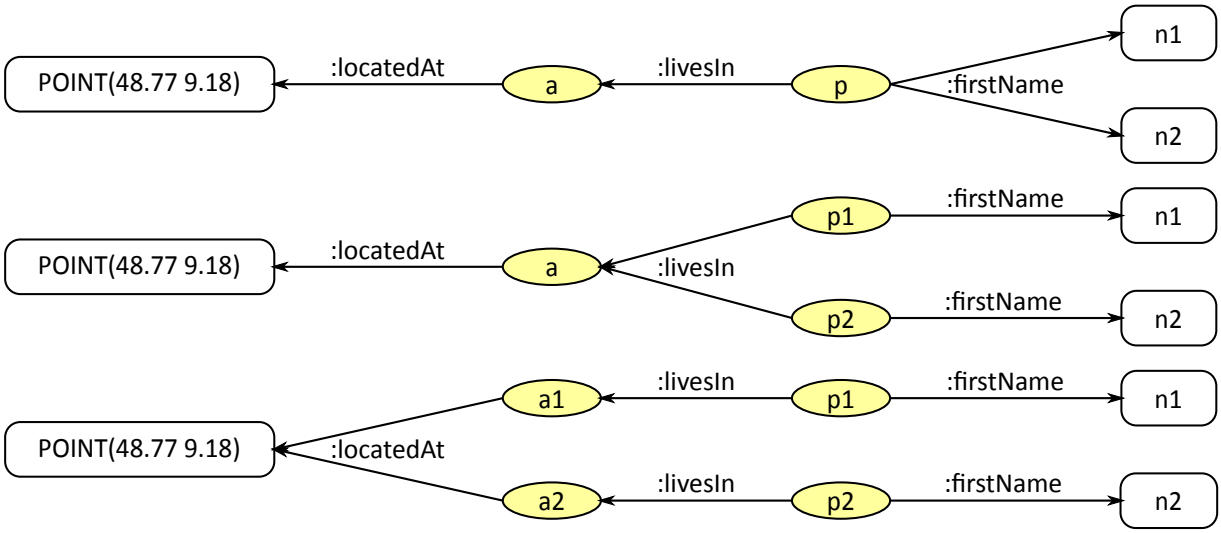


Figure 4.13: Possible RDF subgraphs emulated by path *locatedAt, livesIn, firstName*

Estimating Subgraph Cardinality

The number $card_f$ of features adjacent to a path bundle is an important number for the query optimizer. It is, however, not necessarily the cardinality of the subgraph which the path bundle emulates. If any resource or literal participates in several triples with the same predicate (we hereafter refer to such triples as *multi-edges*), a SPARQL query creates cross-products on all possible combinations. If a query that asks for all occurrences of path π_1 in Listing 4.12 is evaluated on a person with two first names, the resulting cardinality must be two—even if the spatial feature f is involved only once. Figure 4.13 illustrates how a path can have originated from different subgraph structures, if multi-edges are involved. We lost the original subgraph representation by storing only the path bundles. Thus, to emulate the actual subgraph cardinalities, we must emulate the cross-products over multi-edges as well. At this we must differ between queries with variable and constant path ends.

Variable Path Ends If a query contains a path with a variable at the end, the result cardinality includes the cross-product on all combinations of multi-edges. E. g. if the query asks for *all* first names of *all* people living in *any* apartment a at location f, the cardinality is two for any of the subgraph examples of Figure 4.13. Note that for cardinality estimation it is irrelevant which predicate in the path is a multi-edge. For this purpose we only determine the number μ of different resources or literals that are *reachable* from *any* spatial feature f via path π. We store μ together with π.

If a query contains only paths π_i with variable path ends, we determine $card_f$ using Equation 4.1. Then we multiply the result with the number of their reachable end nodes μ_i to estimate the total subgraph cardinality $card_{var}$, as shown in Equation 4.2.

$$card_{var} = \prod_i \mu_i * \left| \bigcap_i F_{\pi_i} \right| \tag{4.2}$$

Constant Path Ends If a query contains a path with a constant value (URI or literal) at its end, it selects this particular value, e. g. the first name "Harry". As we only store the triple predicates, we neither know which nor how many paths ended with this value. So far, all we know is the number F_π of different spatial features adjacent to path π (in the current bucket) and the number μ of different values reachable via π. This tells us that, on average, π connects every spatial feature $f \in F_\pi$ to $\frac{\mu}{|F_\pi|}$ different values. If we assume that the constant values are distributed equally, then a constant value at the end of a path reduces cardinality by the factor $\sigma = \frac{|F_\pi|}{\mu}$. This overestimates cardinality if the constants are not equally distributed. E. g. some names are less common than others and thus more selective than "average" names. Yet, we do not expect severe drawbacks from this approximation.

It is important to observe that constant path ends may be statistically dependent. E. g. the first name "Harry" implies the value "male" for the attribute *gender*. Also Neumann and Moerkotte [2011] observed that one constant value in a Characteristic Set tends to define others, thus acting like a candidate key. Thus, if a query contains several paths with constant ends, we only consider the smallest factor $\sigma_{min} = min(\sigma_j)$ of all paths π_j with constant ends in the query. If the query contains only constant path ends, we consequently approximate cardinality $card_{const}$ as shown in Equation 4.3.

$$card_{const} = \sigma_{min} * \left| \bigcap_j F_{\pi_j} \right| \tag{4.3}$$

Combined Subgraph Cardinality To calculate the cardinality of queries of arbitrary path ends, we need to split all paths of the query into the set P_V with variable ends and P_C with constant ends. The number $card_f$ of adjacent spatial features is calculated using the union of both path sets as shown in Equation 4.1. Then cardinality is adjusted to cater for the variable and constant path ends, combining the approaches

of Equations 4.2 and 4.3. We thus estimate the combined subgraph cardinality as shown in Equation 4.4.

$$card = \sigma_{min} * \prod_{\pi \in P_V} \mu_\pi * \left| \bigcap_{\pi \in P_V \cup P_C} F_\pi \right| \tag{4.4}$$

Pruning Infrequent Paths

The total number of paths adjacent to any spatial feature grows exponentially with path length d. On the other hand, the query optimizer does not need the exact cardinality, as long as an estimate suffices to compare alternative execution plans. Therefore, it is sufficient to keep track of those paths that occur frequently and thus lead to *large* cardinality estimates. Infrequent paths can be dropped; and if a path cannot be found in the statistics, a qualitative statements, such as "few", is enough for the optimizer. Thus, we apply a threshold below which we discard a path. We record a default value that states how high the cardinality of a dropped path can be at most. If we cannot find an entry for a path, we use this default value instead.

There are many possibilities to define such a threshold. A *constant number* of paths for every bucket disregards spatial skew and penalizes buckets with large numbers of different paths. For this reason we put the threshold at a *constant fraction* c of paths for every bucket. This grants buckets with many different paths more storage space and also addresses spatial skew.

Another question is on what measure to apply the threshold. Reasonable candidates are the number $\left|F_\pi\right|$ of spatial features at the beginning of a path π, the number μ of different values reachable via π, or a possibly weighted combination of both. We only apply $\left|F_\pi\right|$, as paths adjacent to very few spatial features of a bucket do not play an important role in the *spatial* distribution, even if they lead to many different values due to a massive amount of multi-edges. The non-spatial RDF statistics capture these multi-edges as well and we can combine their estimate with the default value.

Thus, to reduce the total number of paths recorded for each bucket, we sort all paths by the number $\left|F_\pi\right|$ of spatial features at their beginning. Then we keep the topmost $c * 100\,\%$ ($c \in [0 \ldots 1]$) of the paths and discard the rest. In addition, we keep the number $\left|F_{\pi_\theta}\right|$ of spatial features adjacent to the first path π_θ that was not kept. $\left|F_{\pi_\theta}\right|$ serves as the default value to use if a path π_ϵ cannot be found—π_ϵ cannot correspond to more than $\left|F_{\pi_\theta}\right|$ spatial features. In our experiments we found that a value of $c = 0.3$ to 0.4 still returned reasonable results.

```
...   ?x :name ?name.  ?y :name ?name.  ...
```

Listing 4.13: Triple patterns creating a join over a literal value

Pruning Extremely Frequent Paths

We collect the path statistics using a breadth-first search across all buckets. We start with a sorted list of the spatial features and their buckets. We join the list repeatedly with the *OPS* and *SPO* triple indexes to follow adjacent incoming and outgoing triples, respectively.[8] At this, we do not differ between dictionary IDs of resources and literals. I. e., a path may lead from a resource to a literal value via an outgoing triple and continue from the literal via an incoming triple. For very frequent literal values this causes tremendous occurrence counts of the same path over and over. In our evaluation, for instance, all resources carrying a spatial feature were annotated with *visible*=*"true"*. Thus, the path $\pi_{☠} = (locatedAt \rightarrow, visible \rightarrow, visible \leftarrow, locatedAt \leftarrow)$ connected all spatial features with each other, which resulted in about $8 * 10^{13}$ total occurrences of $\pi_{☠}$ and 9 million occurrences per spatial feature. In the final synopsis this is not a problem, as it only leads to a large value of μ. However, it renders computing the synopsis impossible in practice.

One way to deal with this problem would be not to follow paths behind literals. The dictionary knows whether an ID represents a resource or a literal so this could be easily checked. However, only paths which include literals may capture joins over values. Also, it is not obvious which variable in a query corresponds to a literal, such as *?name* in the triple patterns of Listing 4.13. Finally, paths which occur extremely often at the same spatial features are of course not restricted to literals, even if we did not meet such cases in our experiments.

We chose to address extremely frequent paths by reducing the search depth d dynamically. To determine the value for μ it is not necessary to follow a path μ times. This value is also available from the aggregated triple indexes (see Section 3.1.3). Thus, in addition to joining the temporary path list with the *OPS* and *SPO* indexes, we also scan through the aggregated *OP** and *SP** indexes. The aggregated indexes return the number of reachable resources or literals in the next step of the path. We use it to compute μ. Only below a defined threshold we follow the path any further. Otherwise, we mark the path as pruned. This is required so that the query optimizer can later differ between an extremely frequent pruned path and an infrequent dropped path.

[8] "Incoming" and "outgoing" are defined as seen from the spatial feature along the path.

h [1000]	2	8	32	128	512	2 048	8 192
bucket width [km]	16.76	8.39	4.20	2.10	1.50	0.52	0.26
avg. no. features per bucket	370 000	92 500	23 125	5 781	1 445	361	90

Table 4.2: Numbers of spatial histogram buckets h used in the evaluation and resulting width and average number of spatial features per bucket

4.5.3 Evaluation

We evaluated our cardinality estimation approach on real-world data. We used the data about Germany from OpenStreetMap (OSM) and converted it from the XML dump to RDF. We imported the Nodes, which represent points in OSM, together with their tags and metadata, such as author or timestamp. To create a more densely connected RDF graph, we also included ways (i. e. linestrings) and relations. Different to our evaluation in Section 4.4, we did not convert the ways into WKT linestrings, but modelled them explicitly as RDF resources. Also, we created a *place resource* for every geographic point (see Section 4.2.1) to achieve longer paths. This resulted in a test database of 1254 million triples that contained about 86.6 million spatial features and occupied about 82.8 GB of storage space without the path bundle synopsis.

We created spatial histogram buckets following the cell density approach of PostGIS, as explained in Section 4.5.1. We did this for simplicity, but it is important to note that the path bundle synopsis works with any, possibly more sophisticated, histogram technique as long as it creates histogram buckets. We created several versions of the path bundle synopsis: We varied the total number of created histogram buckets h. Table 4.2 shows the applied values for h. It also lists the bucket width and the average number of spatial features per bucket that result from each value of h. Also we varied the threshold for infrequent paths $c \in \{0.1, 0.2, \ldots, 1\}$. Depending on the number of histogram buckets used, the uncompressed path bundle synopsis occupied between 2.3 % and 10.7 % of additional space. This is a lot, however, for testing purposes the synopsis contained a lot more information than needed by the query optimizer. Also, we could achieve considerable space improvements using compression techniques.

We designed a number of test queries with different selectivities of the spatial and non-spatial parts and different path lengths. We measured the estimated cardinality applying different numbers of histogram buckets h, different threshold for infrequent paths c and considering paths up to maximal length $d = 2$ and $d = 3$. We visualize most results in a three-dimensional figure, plotting the estimated cardinality of a fixed d over h and c. The actual cardinality is a horizontal plane in the three-dimensional

space; we visualize it indirectly using red coloring for over-estimations, blue for under-estimations, and yellow for accurate estimations.

Selective Spatial Filters

The first two test queries have both a selective spatial query predicate and a selective RDF graph pattern. As shown in Listing 4.14, query 1 also includes a constant path end, while query 2 contains only variable path ends, which makes it slightly less selective. Query 1 searches places for leisure activities which were recorded by OSM user *pschaefer* in the city center of Stuttgart. Query 2 retrieves those recorded by *any* user. The actual result cardinality of the queries is *one* and *seven*, respectively.

Figure 4.14 shows the estimated cardinality for the two queries. Large buckets (i. e. small h) tend to underestimate, whereas small buckets overestimate, especially for selective thresholds for infrequent paths (i. e. small c). This is because the estimate for each bucket is considered according to the fraction by which it overlaps with the query region, thus assuming equal distribution inside the buckets. For larger buckets this interpolation may lead to underestimations (and even cardinality estimations below 1), as larger buckets are less likely to show equal distribution of spatial features for real-world data. We will investigate this effect in further detail in the discussion of queries 6 and 7.

Contrarily, the overestimations for smaller buckes are caused by the conservative default value that is applied when a path π is not found in the synopsis: we use the maximal upper bound in order not to underestimate. Yet, if π does not occur in a bucket at all, this value is used, too. For many small buckets, the cardinality of π is overestimated more often than for fewer larger buckets. Longer paths ($d = 3$) show this effect quite dramatically. For shorter paths ($d = 2$) it occurs as well, but is mitigated by the considerably lower number of total paths, which makes it more likely that all paths are found in the synopsis.

The constant path end of query 1 makes quite a difference to the variable path ends of query 2. The threshold c affects the estimates for query 1 with path length $d = 2$ a lot more than for query 2, as the most selective path is discarded. Generally speaking, for moderate values of h and c, such as $h = 32\,000$ and $c = 0.3$, these effects neutralize each other and cause quite accurate estimates.

Unselective Spatial Filters

Queries 3 and 4 represent the class of queries with an unselective spatial filter but with a selective RDF triple pattern. Query 3 includes a constant path end, while query 4 contains only variable path ends, as shown in Listing 4.15. They return all

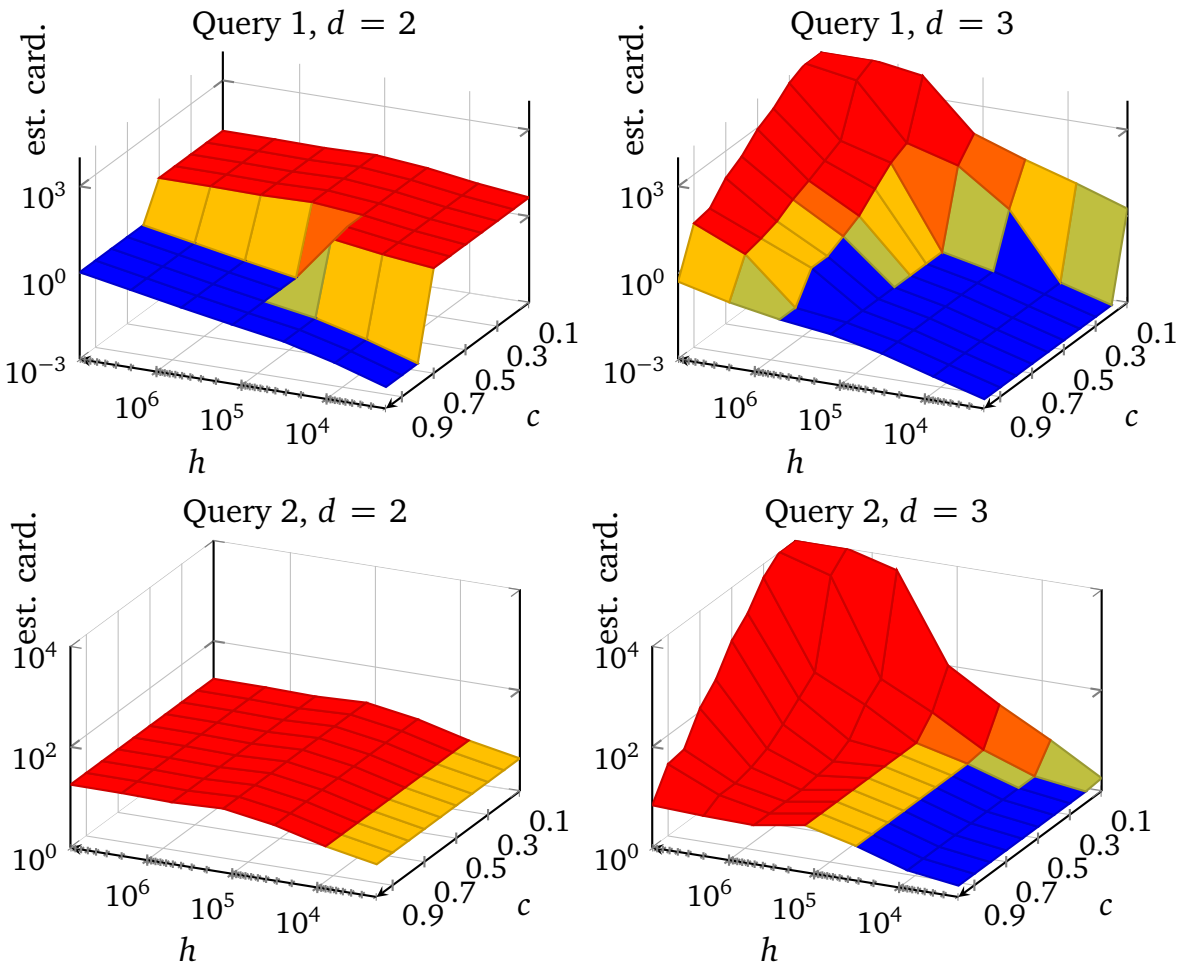


Figure 4.14: Estimated cardinality for queries 1 and 2 (over-estimations marked red, under-estimations marked blue)

```
SELECT * WHERE {
    ?node   :leisure    ?type.
    ?node   :locatedAt  ?point.

    ?point  :user       "pschaefer".  # Query 1 only
    ?point  :user       ?user.        # Query 2 only

    ?point  :coords     ?coords.
     FILTER geordf:within (?coords,   # Stuttgart, downtown
        "POLYGON((9.165 48.774, 9.185 48.774, 9.185 48.785,
        9.165 48.785, 9.165 48.774))"^^geordf:geography)
}
```

Listing 4.14: Queries 1 and 2: Selective spatial filter

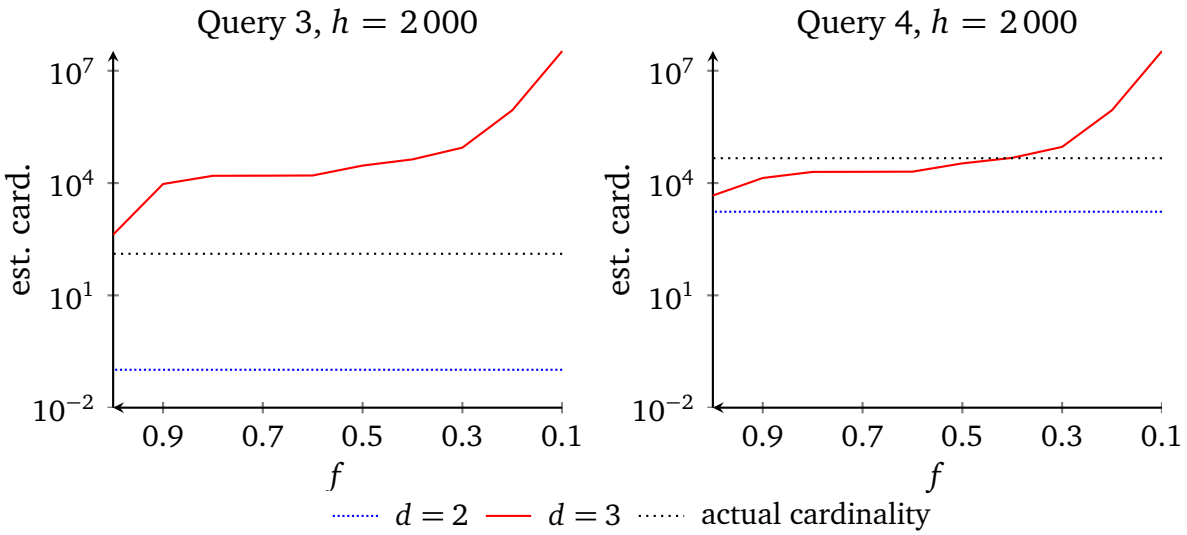


Figure 4.15: Estimated cardinality for queries 3 and 4

```
SELECT * WHERE {
    ?restaurant  :name        "Zur Linde".    # Query 3 only
    ?restaurant  :name        ?name.          # Query 4 only
    ?restaurant  :amenity     "restaurant".
    ?restaurant  :locatedAt   ?point.
    ?point       :coords      ?coords.
     FILTER geordf:within (?coords,           # Germany
        "POLYGON((6 48, 14 48, 14 55, 6 55, 6 48))"^^geordf:geography)
}
```

Listing 4.15: Queries 3 and 4: Unselective spatial filter

restaurants in Germany. Query 3 further restricts the result set by mandating the restaurants to be named *"Zur Linde"*. This name is not uncommon in Germany, but nevertheless cuts down the results considerably. The actual result cardinalities are 129 for query 3 and 46 128 for query 4.

Figure 4.15 shows the cardinality for queries 3 and 4 that was estimated using maximal path lengths d of 2 and 3 and dividing the geographical space in $h = 2\,000$ buckets. Interestingly, the estimates using $d = 2$ are quite different even though the two queries are identical paths up to path length 2. This is due to the non-spatial cardinality estimation, which takes over when the maximal path length of the spatial synopsis is exceeded. We used the cardinality estimation technique of Neumann and Weikum [2009], which tends to underestimate. For path length $d = 3$ the estimations are somewhat similar to those observed for Queries 2 and 3 (which query a much

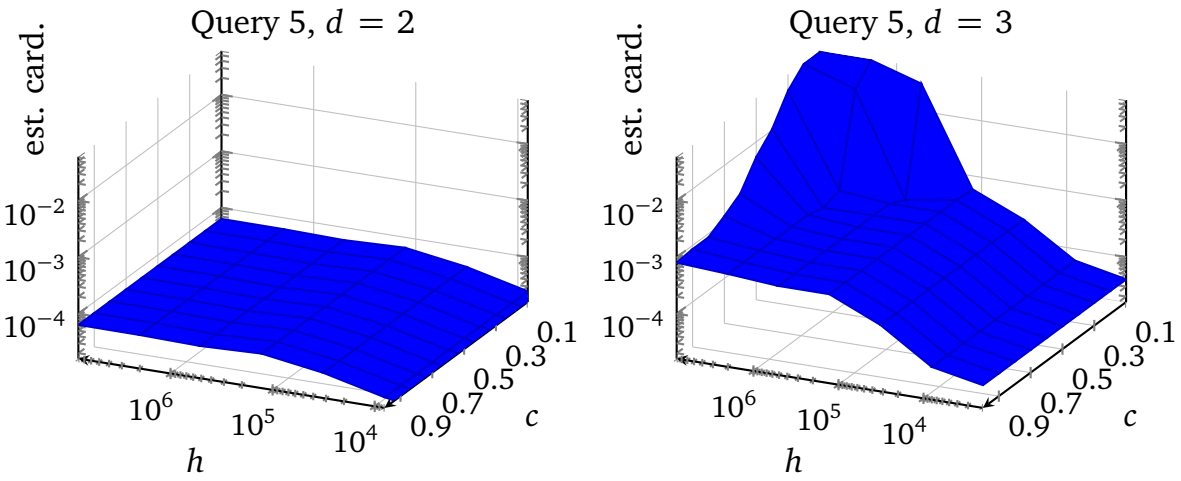


Figure 4.16: Estimated cardinality for query 5 (blue color indicates under-estimation)

```
SELECT ?routeName ?stationName WHERE {
    ?route    :route            "subway".
    ?route    :name             ?routeName.
    ?route    :unspecifiedRole  ?station.
    ?station  :name             ?stationName.
    ?station  :locatedAt        ?point.
    ?point    :coords           ?coords.
     FILTER geordf:within (?coords,            # Stuttgart, downtown
        "POLYGON((9.165 48.774, 9.185 48.774, 9.185 48.785,
        9.165 48.785, 9.165 48.774))"^^geordf:geography)
}
```

Listing 4.16: Query 5: Deep RDF graph pattern

smaller spatial region) with larger numbers of buckets h. Thus, the estimation error does not depend on the number of histogram buckets into which the database is divided, but on the number of buckets that cover the query region.

Deep RDF Graph Patterns

Query 5 includes paths that are longer (4) than the maximal path length recorded in the spatial synopsis ($d = 3$ or 3). This forces the estimates to incorporate information from the non-spatial RDF synopsis. In our measurements we used the cardinality estimation technique of Neumann and Weikum [2009], which, despite all sophistication, suffers from assuming statistical independence. We observed that it tends to underestimate by orders of magnitude.

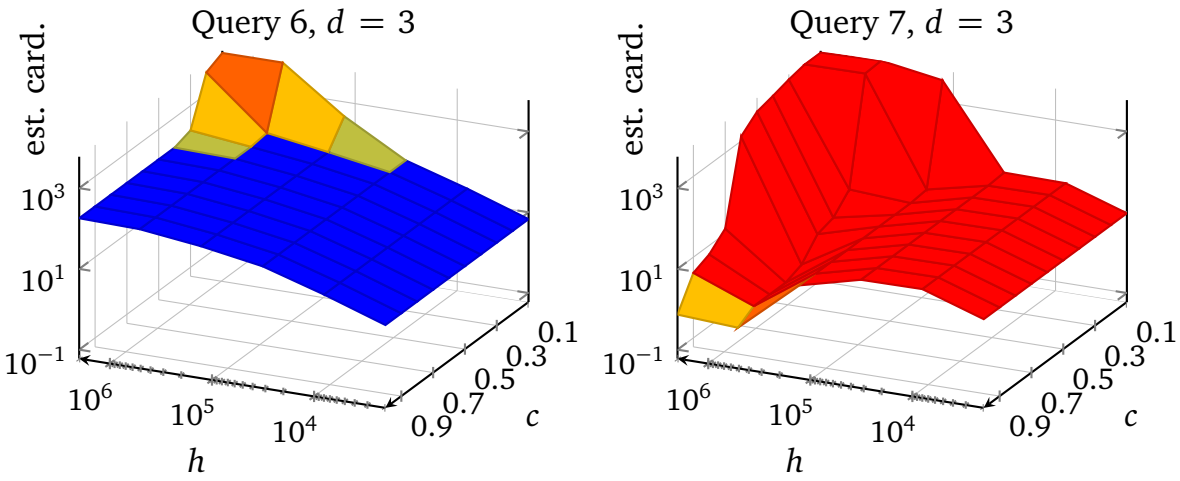


Figure 4.17: Estimated cardinality for queries 6 and 7 (over-estimations marked red, under-estimations marked blue)

As shown in Listing 4.16, query 5 retrieves all subway stations and their adjacent subway routes in downtown Stuttgart. The actual cardinality of query 5 is 69.

Figure 4.16 shows the estimated cardinalities. For maximal path length $d = 2$ the non-spatial synopsis totally dominates the estimates. The number of buckets h and the threshold c hardly influence the results; and the underestimation of the non-spatial synopsis results in total estimates around 10^{-4}. The estimates for $d = 3$ show the effects experienced with the spatial synopsis for the other queries. However, also for $d = 3$ the non-spatial synopsis causes the overall result to underestimate considerably.

Under- and Overestimations for Large Histogram Buckets

Our approach divides the geographic space into histogram buckets and assumes uniformity inside each bucket. It interpolates when a query region covers a bucket only partly. Thus, it under- or overestimates when a part of a bucket is queried that contains more or less than average relevant features, respectively. To measure this effect, queries 6 and 7 search for all shops and the sold products in a shopping street (query 6) and in a city park (query 7) in Berlin, as shown in Listing 4.17. In the shopping street there are actually 207 shops, whereas the park contains only one.

Figure 4.17 shows the estimated cardinalities measured with maximal path length $d = 3$. For small histogram buckets (i. e. large h) the results are comparable to those of the other queries. Yet with growing bucket size, the estimates for query 6 go

```
SELECT ?name ?product WHERE {
    ?shop   :shop       ?product.
    ?shop   :name       ?name.
    ?shop   :locatedAt  ?point.
    ?point  :coords     ?coords.
     FILTER geordf:within (?coords, # Query 6 only: Berlin, shopping street
        "POLYGON((13.315 52.50, 13.33 52.50, 13.33 52.51,
        13.315 52.51, 13.315 52.50))"^^geordf:geography)
     FILTER geordf:within (?coords, # Query 7 only: Berlin, city park
        "POLYGON((13.35 52.51, 13.365 52.51, 13.365 52.52,
        13.35 52.52, 13.35 52.51))"^^geordf:geography)
}
```

Listing 4.17: Queries 6 and 7: Under- and overestimations for large histogram buckets

down whereas the estimates for query 7 increase. Below $h = 32\,000$ the threshold for infrequent paths c is dominated by the interpolation across the histogram buckets.

4.6 Summary and Outlook

Interoperability with spatial data is a central requirement of our architecture, as most of the data on a mobile device possess spatial relevance of some kind. In this chapter we proposed an approach to integrate spatial query processing deeply into the RDF data management system. We discussed how spatial data and spatial queries can be modeled in RDF and the SPARQL query language. Our presented solution models spatial features as typed complex literals, and defines spatial predicates as filter functions in SPARQL. Furthermore, we discussed the deep integration of these concepts into RDF triple stores, and presented an implementation of a triple store with spatial functionality. Our evaluation showed that some of the modifications for our deep integration approach do create some extra overhead for queries selecting very large amounts of spatial features, but we observed excellent performance for most common spatial query types.

Moreover, we addressed the problem of cardinality estimation for spatial queries to RDF data. This is an essential prerequisite for a query optimizer of a triple store that supports spatial query processing. Our approach divides the geographic space into two-dimensional histogram buckets and approximates for each bucket the most frequent subgraphs that are connected to a spatial feature in that bucket. We approximate the subgraphs by decomposing them into cycle-free paths up to a maximal length d that start at a spatial feature. We store the most frequent paths

up to a threshold c in the so-called *path bundle synopsis*, which avoids assuming statistical independence to a large degree. Our evaluation on real-world data from OpenStreetMap shows that for moderate choices of c and d as well as for bucket sizes that range in the same order of magnitude as the query regions, our approach is well capable of estimating the result cardinality of spatial queries to RDF data.

Future Work

Our implementation fully supports the spatial query predicates of the OpenGIS Simple Features Specification [Herring, 2006] by means of SPARQL filter functions. Nevertheless, our work still has promising future research directions. Our implemented filter functions currently compare a spatial feature with a constant value. This is sufficient for many application scenarios. Evaluating a spatial relationship between two variables, i. e., a spatial join, is a future research challenge, both for query processing and for cardinality estimation. Moreover, our cardinality estimation technique partitions the entire geographic space and is based on the paths of the RDF graph which occur most frequently. A future optimization may take into account the queries that are actually asked and incorporate only those spatial regions and paths that are relevant for these queries. Finally, it should be noted that our approach to integrate spatial query processing into RDF data management is not limited to spatial data. On the contrary, it can be generalized to any domain that can be expressed in a complex data type, searched by a dedicated index, and partitioned into histogram buckets. A full text index may be a good example.

Up to this point we have dealt with interoperability of data on a mobile device. The next chapter addresses interoperability at the data management level *across* devices in ad-hoc fashion.

5

Ad-hoc Inter-Device Connectivity

The two previous chapters focused on providing interoperability between applications on a mobile device and addressed the data management layer of our architecture. As we stated in Section 1.1.3, interoperability at the data management level is also required across devices. Devices largely differ in their capabilities, so not every device possesses the required sensors to obtain needed information on the current context. Moreover, not every sensor is always operational (e. g. GPS indoors). Furthermore, some devices may have better and more complete data at hand than others, for instance maps, traffic data, or building plans. Finally, collaborative applications work with non-public data of the participating users which is only known to the devices of the particular users.

What would humans do in a situation when they need information on their surroundings? Most people would probably ask somebody near them for help to find out, e. g., what time it is or where the next public transport stop is located. Mobile devices possess the ability to exchange data spontaneously via wireless ad-hoc networking. If mobile devices combine their own context models with (parts of) context models from other devices nearby, they obtain a much richer view of the world and context-aware applications become a lot more useful [Abowd et al., 1999].

This chapter addresses interoperability across devices via wireless ad-hoc networking. First, we propose the concept of ad-hoc smart spaces (Section 5.1), which we published in [Brodt and Cipriani, 2009] together with Sailesh Sathish from Nokia Research Center. We discuss incentives for users to participate in resource sharing via ad-hoc smart spaces, which is a non-technical problem (Section 5.2). We explain

the technical foundations of an ad-hoc smart space implementation and present a middleware architecture which is integrated in our platform architecture of Chapter 2 (Section 5.3). Then, we address the problem of resource discovery in ad-hoc smart spaces. We examine different resource discovery protocols which we simulated in typical scenarios. The simulations show that the chaotic nature of ad-hoc smart spaces is difficult to capture. Yet, we observe that request flooding works well for small networks (≤ 20 devices). For larger networks, more complex discovery structures pay off. Finally, we present two sample applications for ad-hoc smart spaces (Section 5.5) before concluding the chapter (Section 5.6).

5.1 Ad-hoc Smart Spaces

Van Gurp, Prehofer, and di Flora [2008] define a *smart space* as follows:

> *A smart space is a multi-user, multi-device, dynamic interaction environment that enhances a physical space by virtual services. These services enable the participants to interact with each other and other objects in a peer-to-peer way in the smart space.*

This is, apart from peer-to-peer interaction, in line with many works in the domain of pervasive computing and ambient intelligence, such as Microsoft's *EasyLiving* [Brumitt et al., 2000], Hewlett-Packard's *CoolTown* [Kindberg et al., 2002], or Stanford University's *iRoom* [Johanson et al., 2002], to name a few. As the definition states, these works focus on a certain physical space and augment it by sensors and intelligent components that are integrated into the infrastructure and thus restricted to the particular physical space.

To put interoperability across devices into practice, we propose the concept of *ad-hoc smart spaces,* which Sailesh Sathish from Nokia Research Center Tampere, Finland, developed with us. Ad-hoc smart spaces differ from the above definition in that they concentrate on end-user devices and do not aim at augmenting a particular physical space using deployed infrastructure.[1] Instead, they happen *ad-hoc* as devices meet and one or more devices require context data. Ad-hoc smart spaces are societies of devices which cooperate spontaneously via wireless communication. Thus, ad-hoc

[1] Note that this does not exclude the possibility of non-mobile devices or fixed infrastructure to participate in ad-hoc smart spaces. It merely defines the focus, the use case scenarios, and the design rationale behind ad-hoc smart spaces.

smart spaces are autonomous, highly dynamic, complementary to other technologies, and designed for practical consumer-oriented use.

5.1.1 Autonomous

An ad-hoc smart space does neither require nor assume any dedicated infrastructure, but constitutes a self-organizing *device society* which solely relies on the mobile devices themselves. A device society is formed when devices are brought into proximity. The devices discover each other, exchange their resource information, and share their resources between applications running on the respective devices.

5.1.2 Highly Dynamic

As devices move, device societies grow and shrink, split up and merge, so the resources which the devices provide appear and disappear very dynamically. Consequently, applications for ad-hoc smart spaces must constantly adapt to the capabilities that are currently available.

5.1.3 Complementary

Ad-hoc smart spaces do not attempt to replace existing context-aware applications or infrastructure-based smart space environments. On the contrary, they improve existing context-aware applications and can be used in addition to infrastructure-based approaches, as they attempt to provide context information where other approaches have already given up (e. g., when navigating without GPS). As a consequence, ad-hoc smart space implementations must act in the background to improve applications that require resource data. I. e., they must not hinder other activities on the device, e. g., by allocating device features exclusively.

5.1.4 Practical and Consumer-oriented

Ad-hoc smart spaces explicitly aim at resource-rich mobile consumer devices, most notably modern smart phones, that are available today, as opposed to wireless microplatforms, such as, for instance, the Intel Mote® [Nachman et al., 2005]. To put ad-hoc smart spaces into effect, we focus on practical solutions that cater for easy adoption. I. e., implementations should run on consumer devices without additional hardware or drastic changes of the software stack.

Thus, ad-hoc smart spaces strongly emphasize fast adoption and “making it happen”. Infrastructure-based smart spaces require comparatively large initial investments, both financially and in terms of installation and configuration efforts, before

providing new benefits to the end-user. Also, infrastructure-based smart spaces most likely imply *multi-vendor* in addition to *multi-user* and *multi-device* from the upper definition of Van Gurp et al. Most notably, the different vendors produce a wide range of utterly different kinds of devices, including small sensors, entertainment systems, information and communication devices, alarm systems, building technology, and more [Martin, 2011]. This requires either long and cumbersome standardization efforts with questionable chances of success, or leads to vertically integrated systems of few selected devices and vendors. Ad-hoc smart spaces take up a very practical position by restricting the range of devices and the possible use cases. This increases the chances of adoption and thus of creating new value, from a consumer's as well as from a business perspective [Sathish, 2011].

5.2 Incentives for Ad-hoc Smart Spaces

Sharing context information via a wireless link will cost battery power. But unlike to personal communication, where helping somebody will earn at least some degree of gratefulness, wireless communication happens solely between devices. The person who benefits from the received information does not know who shared it and cannot say thank you. So why should a user allow his device to share context data if all he gets is a drained battery?

As discussed at the Mobile Data Management conference in Kansas City in 2010 [Zaslavsky, 2010], there must be an incentive for the device owners to share context data with others near them. For this we distinguish between ad-hoc smart spaces of devices

1. inside the same organizational entity
2. in collaborative scenarios
3. of arbitrary owners.

Examples for ad-hoc smart spaces of devices inside the same organizational entity or even owned by the same user can be found, for instance, in company and office use cases, smart home scenarios, or in phone-to-car communication. They do not require further incentives at all, as both the costs and the benefits occur inside the same organization.

Collaborative scenarios are similar. They may cross organizational boundaries, but still follow a common goal that is in the common interest of all device owners or that is mandated by a superior entity. Examples may include meetings of different company representatives, or communication standards mandated by conceivable

future law, e. g. in traffic scenarios. To achieve this common goal, device users are willing or mandated to take into account the resulting costs. Consequently, collaborative scenarios can do without incentives as well.

Thus, only ad-hoc smart spaces of arbitrary device owners require incentives. For this, one must look at the ad-hoc smart space concept in a wider scope that goes beyond its technical properties (and beyond the scope of this work). Thus, a business- or community-oriented concept must be established. A business-oriented incentive concept could be integrated in other offerings of a device vendor or network operator. For example, users who are willing to help others with context data could earn credit points or discount vouchers which they can spend to purchase other offerings of the company, e. g. in an app or music store. Some online stores apply similar concepts in order to persuade customers to write product ratings. A device vendor might make such an offering because its devices and applications become more useful with more context data at hand, which may be an advantage over a competitor. A network operator may benefit from reduced network load if devices obtain context data directly from their device neighborhood without a roundtrip via server-sided infrastructure.

A community-oriented incentive concept must build up a notion of honor, for instance publish a high-score list of the “most helpful people of the week”. Similar incentive concepts exists, e. g., in volunteer computing [SETI@home Project, 2011].

5.3 Technical Foundations and Architecture

The technical foundations for ad-hoc smart spaces are most notably constituted by Bluetooth networking. It strongly influences the architecture of an ad-hoc smart space middleware and is also a major design driver for resource discovery, as discussed in Section 5.4.

We focus on Bluetooth, as it best meets the requirements *practical* (Section 5.1.4) and *complementary* (Section 5.1.3): Most mobile devices support only Bluetooth and WLAN for local wireless communication. Unlike WLAN, Bluetooth does not interfere with ulterior communication. Contrarily, a WLAN-solution needs to put the WLAN interface into ad-hoc mode. This cuts off all infrastructure-based communication, so that the device looses its connections to the internet via a WLAN access point. Finally, Bluetooth is comparatively energy-efficient, which suits it for long-running background activities, as implied by the *complementary* requirement. Our ultimate goal is a practical implementation on current mobile consumer devices, such as the Nokia Internet Tablets. This further restricts the design space to solutions that are

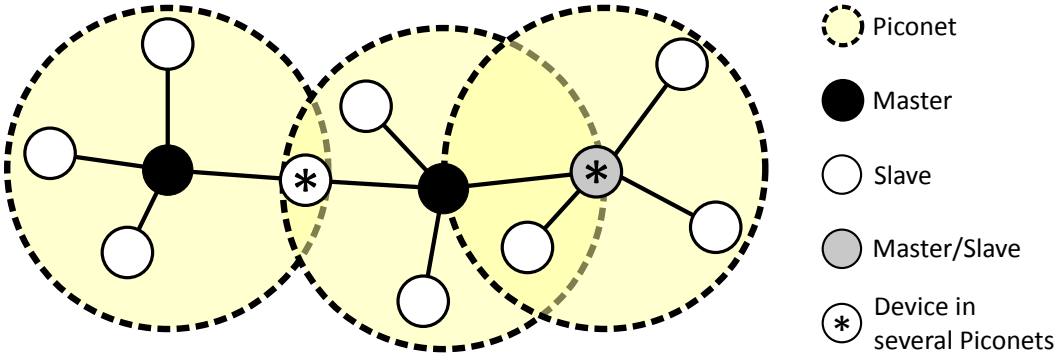


Figure 5.1: A Bluetooth scatternet consisting of three piconets

implementable on top of a state-of-the-art mobile software platform, in our case Linux and the BlueZ Bluetooth stack.

5.3.1 Bluetooth Networking

Bluetooth is a wireless communication standard for data exchange over short distances. It is intended for personal area networks (PANs) of mobile devices and offers a high level of security. It was originally created by Ericsson in 1994 as an alternative to data cables.

Bluetooth communicates strictly in a point-to-point fashion. Devices organize themselves in *piconets* of one master and multiple slaves. The master coordinates communication in the piconet and routes messages between slaves at the physical link layer. The slaves only communicate with each other on the logical link layer.

A device can participate in several piconets at the same time. If the network includes such devices, as shown in Figure 5.1 (marked with an asterisk), the network is called a *scatternet*. The Bluetooth specification [Bluetooth Special Interest Group, 2001] does not include routing in a scatternet, and current Bluetooth stacks do not provide it. Applications need to provide their own scatternet routing on top of the Bluetooth stack.

Bluetooth includes the Service Discovery Protocol (SDP), a mechanism to find *services* in the local piconet [Bluetooth Special Interest Group, 2001]. However, resources further away in a scatternet are out of reach. Also, SDP is restricted to searching for unique service IDs. Search by arbitrary attributes (e. g., “GPS devices currently receiving three-dimensional coordinates”) is not supported.

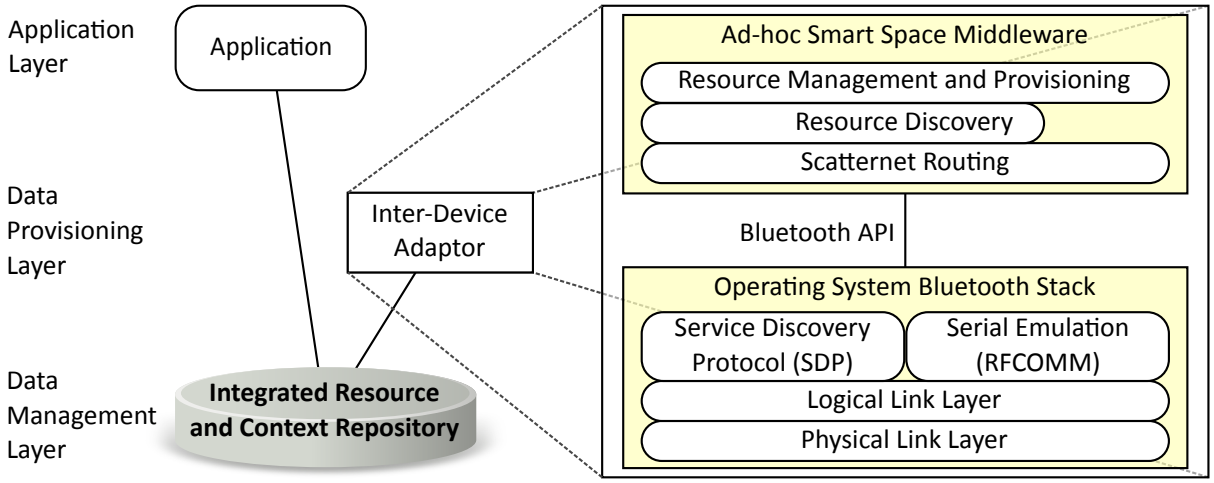


Figure 5.2: Architecture of an Ad-hoc Smart Space implementation and integration into our platform architecture of Figure 2.1

5.3.2 Architecture of an Ad-hoc Smart Space Middleware

Figure 5.2 shows the architecture of an ad-hoc smart space middleware. It is integrated into our platform architecture described in Section 2.2 as the implementation of the "Inter-Device Adapter". The ad-hoc smart space middleware is implemented as one Bluetooth service that builds a peer-to-peer overlay on top of a Bluetooth scatternet. We use Bluetooth device discovery and SDP to find running middleware instances first. Then the middleware instances organize each other to share their resources in a second step.

Put simply, the Bluetooth stack of the operating system comprises three layers. The physical link layer manages master-slave communication. The logical link layer caters for datagram messaging in a piconet and provides the base for the Service Discovery Protocol (SDP) and the Radio Frequency Communication (RFCOMM) profile. RFCOMM emulates a serial port and thus provides reliable end-to-end communication.

The ad-hoc smart space middleware uses RFCOMM and needs to provide scatternet routing. Resource discovery builds on top of the routing layer to distribute resource information throughout the scatternet. Resource management uses both resource discovery and routing. An application is presented a unified set of resources; local and remote.

5.4 Resource Discovery in Bluetooth-based Ad-hoc Smart Spaces

A key challenge of ad-hoc smart spaces is *resource discovery*. Unlike device discovery, which is closely coupled with the underlying communication technology, finding out which devices might share which resources opens a large space. A plethora of discovery mechanisms exist in the literature [Meshkova et al., 2008], ranging from home entertainment to internet-scale peer-to-peer nets. We evaluate how selected resource discovery protocols perform in ad-hoc smart space settings characterized by the following three design drivers, as published in [Brodt et al., 2010b]:

1. Autonomous peer-to-peer organization
2. Highly dynamic behavior
3. Bluetooth communication

Figure 5.2 depicts resource discovery as a core component in our middleware architecture that acts on top of the entire scatternet. Our scenario is characterized by a relatively small scale (at most a few hundred devices), decentralized autonomous organization, and very dynamic network topology. This rules out *centralized* client/server approaches, such as LDAP [Howes and Smith, 1997] or DNS [Mockapetris and Dunlap, 1988].

Decentralized *structured* approaches, including Chord [Stoica et al., 2001] or CAN [Ratnasamy et al., 2001], distribute resource information using sophisticated structures. The effort to maintain these structures in the highly dynamic network is likely prohibitive, so we did not consider these approaches.

Many peer-to-peer networks build clusters as an overlay network for efficiency [Meshkova et al., 2008]. A Bluetooth scatternet provides a cluster structure at the physical link layer, so it seems ideal to exploit it. Greede and O'Mahony [2003] present an algorithm that routes Bluetooth SDP queries through scatternets. Devices participating in several piconets offer an SDP service named "bridge". A device looking for resources may use a bridge service to forward SDP searches to other piconets, if the service was not found in the local piconet. Besides adopting the limitations of Bluetooth SDP (search by service IDs only), there is one major problem, however. A device must determine its role in the scatternet to know whether to provide the "bridge" service. This knowledge is available in the physical link layer (see Figure 5.2), but the Bluetooth API, as provided on a mobile device, does not propagate the knowledge to the logical link layer. In practice, scatternet routing and resource discovery have no chance to determine a bridge device. Of course it is possible to modify the Bluetooth stack of an open operating system. Yet, this would

render easy adoption of ad-hoc smart spaces impossible on a significant number of devices and thus violate the requirement *practical and consumer-oriented* mandated in Section 5.1.4.

Our scenario leaves three main design variables open: replication, routing, and information propagation.

1. The *degree of replication* varies between zero and full replication. The extremes are pull-based querying of all devices or push-based replication on all devices and search in local directories, respectively.

2. Protocols may integrate *scatternet routing* with discovery, or keep the two tasks separate. Integrating them may optimize communication based on the network structure. Separating them allows exchangeable layer implementations and even other communication technologies.

3. Protocols may *propagate information* in different ways: flooding, tree structures, or point-to-point links.

We evaluated resource discovery protocols covering all three design variables. The following sections introduce these protocols.

5.4.1 Request Flooding

Request Flooding uses zero replication, integrates scatternet routing, and propagates information through flooding. A searching device emits a request to all neighbors, which either reply with resource information or forward the request recursively. Unique request IDs prevent multiple processing of requests. Request Flooding does not maintain any support structures or directories and needs no initialization. Yet it is guaranteed to find any existing resource. It also finds the shortest route to the resource. Due to its simplicity, Request Flooding is considered trivial. At the same time it serves as a benchmark; other protocols should perform at least as good as Request Flooding.

5.4.2 Resource Flooding

Resource Flooding is a contrary approach to Request Flooding. It uses full replication, integrates routing and propagates through flooding. Devices publish their resource information to all neighbors, which forward it recursively. This happens whenever resources become available. Devices keep a local directory that is updated by the resource advertisements. Resource requests are answered locally without any communication.

5.4.3 Publish/Subscribe

Publish/Subscribe [Herms and Schulze, 2008] uses full replication, includes routing, and propagates resource advertisements via a tree to avoid the duplicate messages, which Resource Flooding produces. When two devices encounter each other, they exchange their lists of known devices. For every unknown device in the received list, they issue a subscription to receive all resource advertisements about the (previously) unknown device. Thus, devices need to keep a subscription table to memorize whom to forward incoming resource advertisements. The subscriptions form a *tree* for every resource provider, which must be repaired whenever a device leaves. When the loss is noticed, it is published along the tree, and devices remove the respective resources from their local directories. The devices exchange their lists of known devices regularly so that broken paths in the tree are ultimately replaced.

5.4.4 Gnutella-Inspired

Gnutella [Klingberg and Manfredi, 2002] was the first major peer-to-peer file sharing system that worked completely decentralized. Every participant selects a number of *neighbors* anywhere in the network that replied to a flooded ping message. File Search is done by flooding a request over these neighbors (using a max. hop counter). This scaled poorly for very large networks and led to enormous delays. Several improvements were proposed including clustering [Chawathe et al., 2003] using "ultra-peers" that cache resources of their neighbors. As ad-hoc smart spaces scenarios are much smaller than internet-scale file sharing nets, the disadvantages of Gnutella might not make much impact. So we adopted concepts from Gnutella in our "Gnutella-inspired" protocol: A device finds other devices through Bluetooth device discovery and SDP, declares them its neighbors and connects to them. It floods a ping message over all neighbors (using a max. hop counter). Devices reply with the list of resources they offer. The initiating device stores these lists in a local directory together with the route via which the message traveled. Devices routing the messages to their destination update their directories, too. Discovery requests are answered from the local directory. If this does not succeed, the request is flooded over the neighbors. Thus, the protocol uses partial replication and propagates information through flooding. We designed two variants of the protocol which differ in their routing behavior:

Integrated Routing (IR)

This variant directly calls the Bluetooth API and utilizes direct links in the scatternet. Thus, "neighbors" are reachable via one hop and flooded discovery requests travel along the physical scatternet structure.

Separate Routing (SR)

To evaluate the effect of a separate scatternet routing layer, we modified the protocol accordingly. The particular routing algorithm used in our simulations (see Section 5.4.7) keeps the network slim, resulting in a different network topology. The discovery protocol is the same, devices only connect to fewer neighbors and the routing algorithm creates additional messages.

5.4.5 Central Directory

The Central Directory protocol creates a single replication of all resource information, depends on a separate routing layer and uses point-to-point communication. It implements *publish-find-bind* as known, e. g., from UDDI [Clement et al., 2008]. It works very well in static infrastructures, but requires tweaks in ad-hoc smart spaces. Devices vote one device to host a central directory and to answer all discovery requests. Although the term "vote" implies a certain notion of democracy, the first device needing a directory simply appoints itself directory host and informs the others via flooding, which reply with their resource lists. When the directory host becomes unavailable, the first device to notice the loss initiates a new voting procedure. In case of a net merge two directory hosts exist. A net merge always creates a bottleneck at the single bridge device that routes all messages between the two nets. The bridge is not interested in the traffic of discovery requests, so it tells the two directory hosts to exchange their directories. This way, discovery stays in the two old networks, and no directory announcements need to be flooded.

5.4.6 Random Replication

Random Replication features a configurable degree of replication, depends on a separate routing layer, and uses point-to-point communication. It creates many *decentralized* directories randomly. Random Replication builds an overlay structure independent of the physical network and assumes a global view on the scatternet. Every device randomly chooses a number of "neighbors" from the entire ad-hoc smart space. Every device queries its neighbors for their resource lists and stores them in a local directory. Search is first done locally, then the neighbors are queried. Thus, the

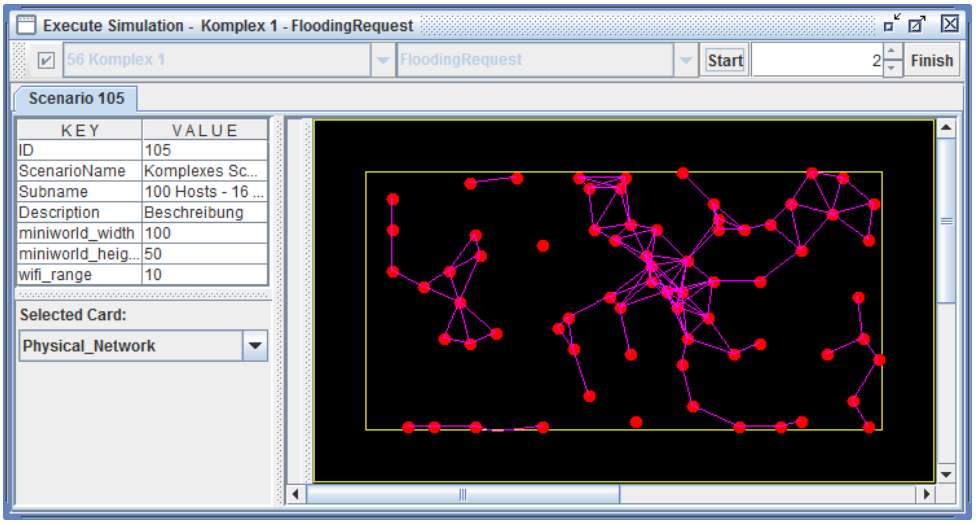


Figure 5.3: Screenshot of the Simulation Environment

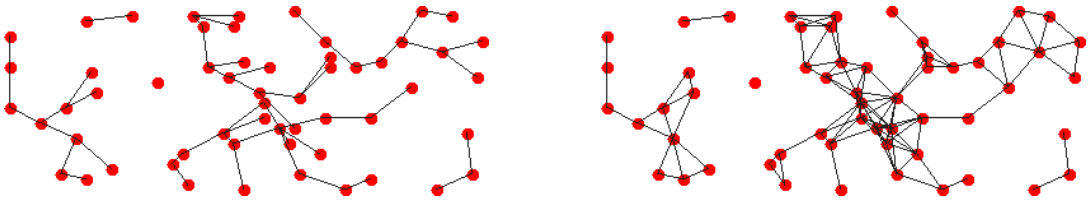
requests may travel quite randomly through the physical network. The parameter r denotes the neighbors per device as a fraction of all devices. Thus, r controls the global degree of replication. The number of logical hops of a request in the overlay network largely depends on r. We simulated $r = 30\%$ and $r = 50\%$. I. e., statistically every third or second request can be answered from the local directory, respectively.

5.4.7 Simulation Environment

We implemented a simulation environment that is specially tailored to evaluate resource discovery protocols in Bluetooth-based ad-hoc smart space settings. The simulation environment is available as open source software [Wobser, 2010]. Figure 5.3 shows a screenshot of the simulation environment.

Functionality

The simulation environment simulates the behavior of a resource discovery protocol by creating the messages that are sent between devices in the Bluetooth piconets in discrete simulation steps. As we are merely interested in the overall behavior of resource discovery protocols that are implemented in higher layers, the simulation environment skips the internal details of the Bluetooth stack. As a “testbed” for the resource discovery protocols, the simulation environment provides an implementation for the components in the system architecture with which a protocol implementation interacts (see Figure 5.2): the scatternet routing layer and the API of the Bluetooth stack.



(a) Sparse network structure based on global knowledge

(b) Dense network structure based on piconet-local knowledge

Figure 5.4: The sparse network structure of the scatternet routing algorithm compared to the dense network created by a greedy approach

The scatternet routing layer allows a resource discovery protocol to send a message to any (simulated) device in the ad-hoc smart space. It provides a list of all devices and is able to list the devices to which a direct link is currently available. Section 5.4.7 explains the routing algorithm which we used in the simulations.

The Bluetooth piconet layer simulates the Service Discovery Protocol (SDP) and piconet messaging. Sending a message in a piconet is performed in a single simulation step, i. e., one piconet message is the smallest measured unit of the simulation environment. The piconet layer emulates the BlueZ API, so that a protocol plugin can be easily turned into a real implementation. Moreover, the piconet layer creates the piconets in the same way as a real Bluetooth stack would. I. e., when devices connect to each other, a realistic network structure emerges automatically.

The simulation environment places the devices in a two-dimensional space and manages their relative positions. Moreover, the vicinity of devices is simulated based on their distance. Devices can be added and removed dynamically during the simulation. We use this to simulate mobility of devices and consider this sufficient to mimic the dynamic behavior of ad-hoc smart spaces. If desired, the simulation environment provides an interface via which complex mobility models can be added as an extension. Figure 5.3 shows how the simulation environment visualizes the positions of the devices and the network topology.

The simulation environment is extensible by plug-ins adding new resource discovery protocols, monitoring plug-ins to collect additional metrics, and scatternet routing algorithms. A database back-end loads the different test cases and stores the test results.

Scatternet Routing

The routing algorithm which the simulation environment implements to provide scatternet routing is based on distance vector routing. Every device keeps a local

routing table which provides a global view on all devices, but not the optimal route. The routing algorithm attempts to minimize the number of open connections per device and to achieve a sparse network topology, as illustrated in Figure 5.4(a). This increases network throughput and decreases message loss, because, in practice, devices require less context switches [Baatz et al., 2002; Cuomo and Pugini, 2005]. The routing algorithm works as follows:

A device periodically searches for other devices that run the ad-hoc smart space service. This is done using Bluetooth device discovery and SDP; in the simulation devices are found based on relative distance of devices. Whenever a device *A* finds a new device *B*, *A* first checks whether *B* is already listed in the routing table. If so, *B* is already reachable somehow, i. e., a direct connection from *A* to *B* would only make the network more dense, as can be seen in Figure 5.4(b). Otherwise *A* connects to *B* and the two devices exchange their routing tables. For every device *D* in *B*'s routing table that is unknown to *A*, *A* inserts a new route into its table with *B* as the next hop and with an incremented hop count. For every device that *A* does know, *A* checks whether the route via *B* is shorter. If so, *A* modifies its routing table accordingly.

If *A* notices that *B* is no longer reachable, it simply removes all routes via *B* from its routing table. Yet, *A* does not propagate the loss of *B* to other devices, as it is not sure whether this information is ever needed. Only if a device attempts to send a message to or via *B*, a notification is sent along the routing path to signal that *B* is gone. Subsequently, all devices on the path remove *B* from their routing tables.

Metrics

The simulation environment exclusively measures the number of messages a device sends and receives on the logical link layer in a piconet. We distinguish between messages that were sent to accomplish a resource discovery task and messages created by the routing algorithm. In both cases we count the messages that a device sends and that it receives.

As stated above, the simulation environment does not simulate the low level details of Bluetooth networking. The smallest simulated unit is one piconet message; message fragmentation, frequency hopping, or TDMA slots are omitted. This rules out measuring precise latencies and exact energy consumption. However, every resource discovery protocol discussed in this paper acts on top of the RFCOMM Bluetooth profile and thus does messaging in the exact same way. I. e., a precise simulation of the low level details would not reveal any additional information to compare the protocols. The number of piconet messages allows estimating latency and energy consumption of a protocol; as they will be roughly a function of the message load.

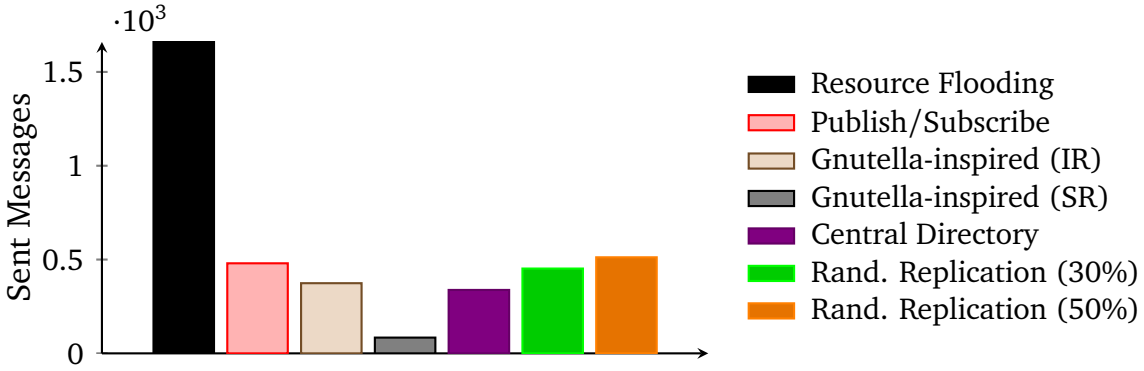


Figure 5.5: Message load measured in the *Initialization* scenario

5.4.8 Evaluation

We compared the resource discovery protocols in over 2 000 simulations. We simulated three network topologies (Line, Circle, and Random), and three network sizes: Small, Medium, and Large (5, 15, and 50 devices).

In the following, we present five selected test suites: *Initialization* measures the initial overhead to build up the organizational structures of the protocols. *Grow* adds new devices to the network continuously to determine the effort of a protocol to adapt to new nodes and to scale up. *Search* tests the message load a protocol produces to find a resource. *Resource Update* adds, updates, and removes resources from devices. Finally, *Random Actions* performs a randomly chosen suite of 30 actions to simulate the complex happenings that are to be expected in a real world setting.

Initialization Scenario

For small networks one would assume that the message load to maintain organizational structures does not pay off. Thus, it is interesting to measure this load as a first performance indicator. We created a test scenario using a *Small–Circle* and a *Small–Line* topology, and no discovery requests. We ran the simulation for the first 20 messaging cycles. Thus, all messages serve the purpose to find neighbors, establish routing, create local directories, etc.

Figure 5.5 shows the results of the initialization scenario. There are two extremes: First, Resource Flooding creates an excessive amount of messages. This is not a specific outlier of the particular scenario; we obtained similar results from other simulations (omitted for brevity). Second, Request Flooding is not shown in Figure 5.5, as it creates no messages at all. This is obvious, as Request Flooding only communicates to search for resources, thus, it does not build up any auxiliary structures.

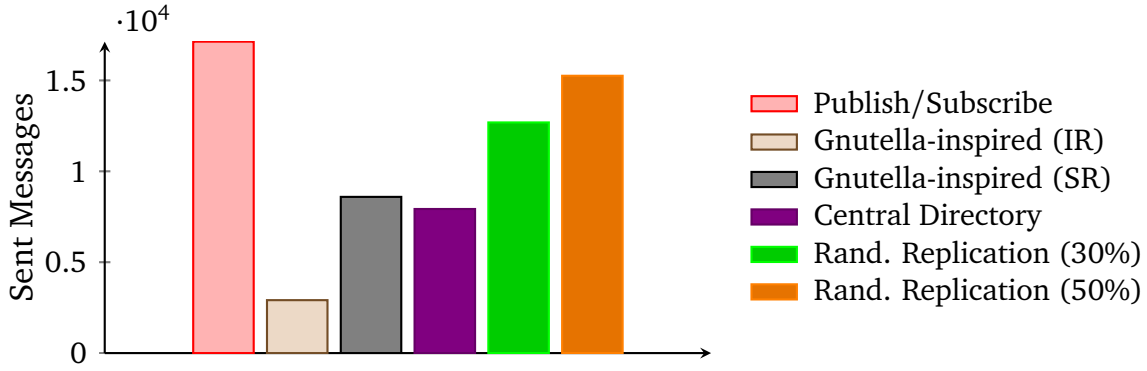


Figure 5.6: Message load measured in the *Grow* scenario

Furthermore, Figure 5.5 shows that the Gnutella-inspired protocol with integrated routing *(IR)* generates considerably fewer messages than with separate routing *(SR)*. Initialization of the routing layer clearly leaves its mark.

Grow Scenario

To test the scalability of the resource discovery protocols, we simulated three network topologies growing incrementally from one to 50 devices. As in the Initialization scenario, there were no discovery requests, communication happens only to establish organizational structures.

Figure 5.6 shows the results summed up for all three topologies, except for the flooding protocols. Resource Flooding created such an excessive message load that we decided not to consider the protocol in our simulation any further. As in the Initialization scenario, Request Flooding does not communicate at all.

Even though the tree propagation of Publish/Subscribe improves Resource Flooding considerably, it still shows the highest message load. The Gnutella-inspired protocol performs very well with integrated routing (IR) and mediocre otherwise (SR), due to the establishment of routing structures. The Central Directory performs slightly better than Gnutella SR. Both rely on the established routing structures, but creating the Central Directory obviously requires fewer messages than flooding pings. Random Replication nicely shows the effect of the parameter r, denoting the amount of neighbors per device. $r = 50\%$ generated nearly 3000 more messages than $r = 30\%$, or 20 more messages in average per device in each topology.

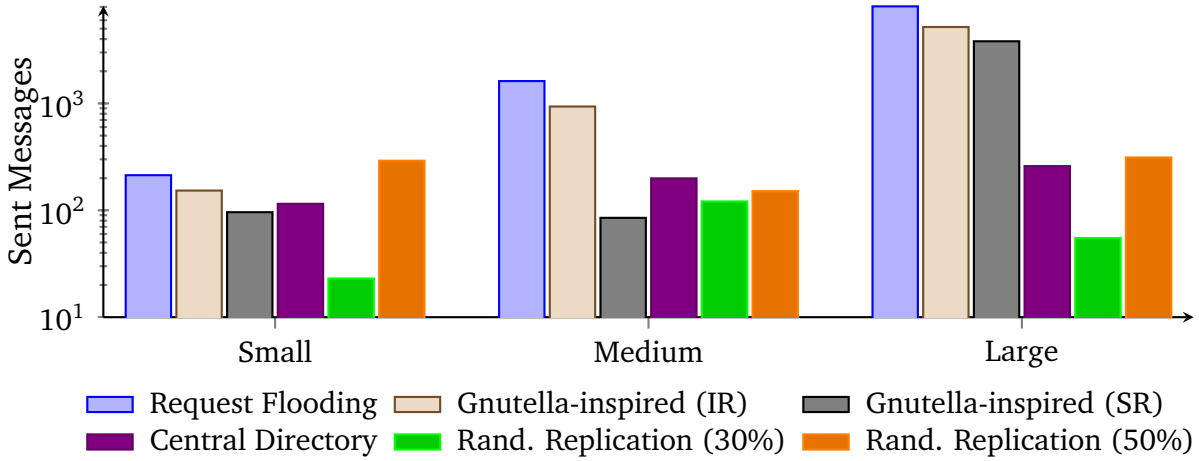


Figure 5.7: Message load measured in the *Search* scenario

Search Scenario

The Search scenario consists of numerous simulations that search for resources in various network topologies. Figure 5.7 shows the combined results grouped by network size. Publish/Subscribe fully replicates all knowledge locally, so devices do not need to communicate at all to find resources. This advantage was bought at the price of very high message load in the Initialization and Grow scenarios. Thus, the Publish/Subscribe protocol did not send a single message in the search scenario and is therefore not included in the figure.

The logarithmic scale of Figure 5.7 nicely shows the exponentially growing message load of Request Flooding. This is unacceptable in large networks. Yet for small networks, Request Flooding performs well. The message load of the Gnutella-inspired protocol is directly proportional to the network size for all network sizes tested. We conclude that the max. hop counter of the tested protocols was too high and thus never came into effect. There is, however, an observable advantage of the separate routing layer (SR) as compared to integrated routing (IR). Thus, the higher initial effort pays off here. The Central Directory achieves mediocre results which are slightly better than Random Replication with $r = 50\%$. The measures for Random Replication do not increase significantly with network size. A value of $r = 30\%$ constantly beats $r = 50\%$. On the other hand the randomness in the protocol makes the Random Replication difficult to predict.

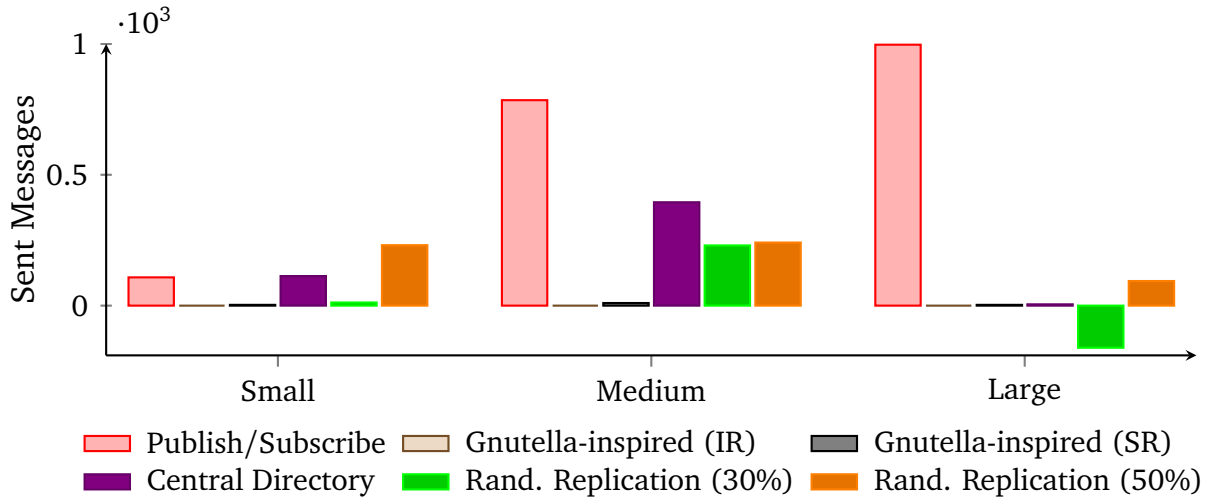


Figure 5.8: Message load measured in the *Resource Update* scenario

Resource Update Scenario

Not only the network topology of an ad-hoc smart space changes constantly, the resources of devices may also come, change and go dynamically. We simulated the message load caused by resource updates in three network topologies having three different sizes. We conducted one simulation for adding, updating, and deleting a resource each, resulting in a total of 29 simulations. To determine the load of the resource updates without maintenance of support structures, we subtracted the load of an empty simulation. Again, Request Flooding did not communicate at all in the simulation and is thus not included in the figure.

The simulated message loads of the Resource Update scenario, as shown in Figure 5.8, are directly related to the degree of replication of the protocols. Publish/-Subscribe, which fully replicates the global resource list on each device, produces the highest message load. The Gnutella-inspired protocols do replicate resource information, but do not actively update the directories. Instead, they exploit resource listings contained in messages they forward for others, thus saving explicit update communication. The Central Directory protocol keeps a single (or very few) replications of the global resource list causing some update load, but considerably less than Publish/Subscribe. Even though the replication rate of the Random Replication protocol is relatively high (depending on the parameter r), it requires comparatively few messages. As expected, $r = 30\%$ caused more update messages than $r = 50\%$.

Figure 5.8 also illustrates the randomness of Random Replication. The medium-sized network produced more messages than the large network, as devices happened to choose different neighbors. Also, the empty test run occasionally caused more

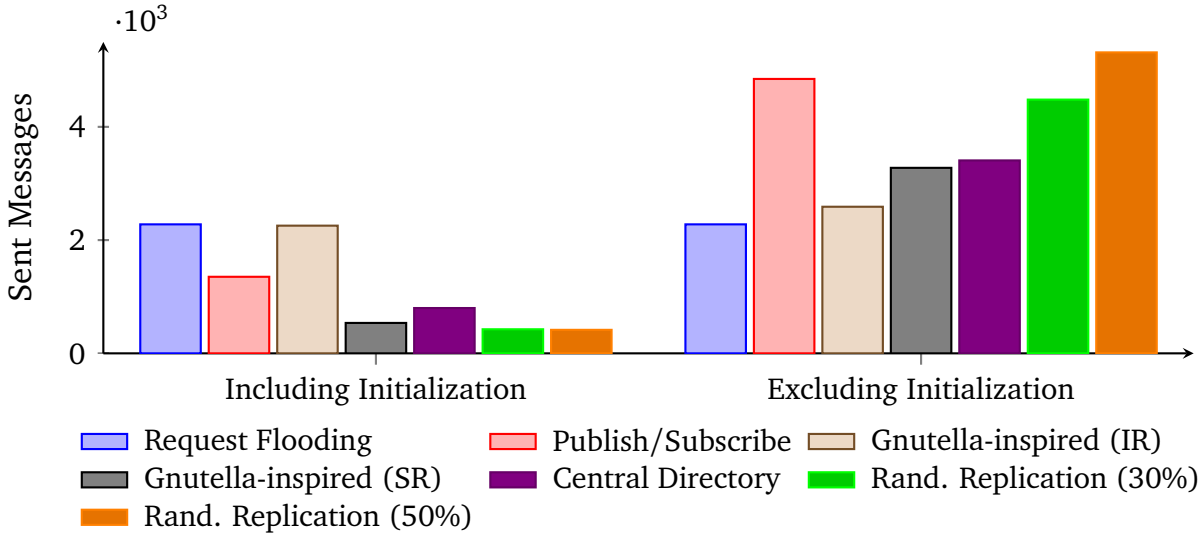


Figure 5.9: Message load measured in the *Random Actions* scenario

messages than the runs that actually simulated the update scenario. This explains the negative value for $r = 30\%$ in the large network.

Random Actions Scenario

To gain a more realistic picture of the protocols we attempted to simulate a real-world scenario by applying a randomly chosen (but constant) scenario of 30 actions in a network of 50 devices and different topologies. Figure 5.9 shows the resulting message load, both excluding and including network initialization.

Excluding initialization, all approaches that separate routing perform better: Random Replication, Gnutella-inspired (SR), and the Central Directory. Publish/Subscribe produces more messages as it keeps all replications up to date. Request Flooding and Gnutella-inspires (IR) produce a lot more messages, as they are based on a denser network and replicate less. They perform almost identically, as Gnutella-inspired (IR) is similar to flooding, if the network is smaller than the max. hop counter.

If we do consider the initialization effort, the figures change. Now, Request Flooding and Gnutella-inspired (IR) perform best, as they neither require routing nor replicate much. Gnutella-inspired (SR) and Central Directory follow; they establish routing, but their replication effort is low. Random Replication and Publish/Subscribe need significantly more messages, as their degree of replication is higher.

	replication	routing	propagation	Initialization	Grow	Search	Res. Update	Rand. Actions
Request Flooding	none	N	flooding	++	++	–	++	∘
Resource Flooding	full	N	flooding	☠☠				
Publish/Subscribe	full	N	tree	+	–	++	––	–
Gnutella-inspired IR	small	N	flooding	+	++	–	++	–
Gnutella-inspired SR	small	Y	flooding	+	∘	–	++	+
Central Directory	once	Y	peer to peer	+	∘	+	–	+
Random Replication 30%	30%	Y	peer to peer	+	–	+	+	+
Random Replication 50%	50%	Y	peer to peer	+	–	+	+	+

Table 5.1: Comparison of the simulated resource discovery protocols

Summary

The results differ considerably depending on the simulated network size, update and search rate. We observed that routing structures and replication do not pay off for small networks. On the other hand, the exponential growth of flooding does not make much impact for small networks. For medium to large networks the effect of support structures becomes visible and clearly helps the respective protocols to scale, as observed with Random Replication and the Central Directory. Due to its robustness, we prefer Random Replication for medium and large ad-hoc smart spaces. Its randomness can make its performance hard to predict, but it also helps the protocol cope with changes in the network. The high message load of Publish/Subscribe in presence of updates shows that full replication of resource information does not pay off, even though search is local. Table 5.1 summarizes the protocols with their characteristics and simulation results.

5.5 Sample Ad-hoc Smart Space Applications

To demonstrate the concept of ad-hoc smart spaces we implemented two sample applications featuring different interaction patterns. First, we put the aforementioned GPS sharing use case into practice, which is also published in [Brodt and Cipriani, 2009]. Second we created the Spontaneous Team Meeting Solution (STEAMS) [Reitschuster,

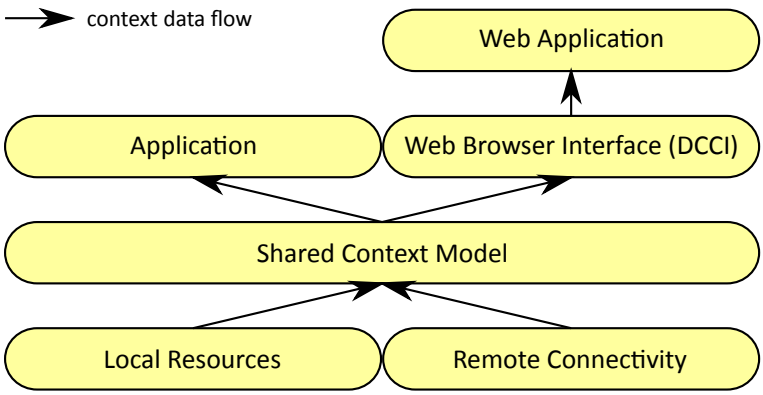


Figure 5.10: Architecture of the GPS sharing demo

2010]. STEAMS is a collaborative application to schedule meetings automatically. The GPS sharing demo uses event-based interaction, whereas STEAMS accesses remote resources in a get/put fashion. Both applications were implemented for Nokia N800 and N810 Internet Tablets running Maemo linux. They were written partly in Python and partly in C++ and utilize D-BUS for local inter-process communication and the pyBlueZ Bluetooth library to communicate with remote devices.

5.5.1 GPS Sharing Demo

Our first ad-hoc smart space demo implements the scenario of mobile devices sharing context data streams, most notably GPS position streams, with others near them. It abstracts from local device capabilities and offers a unified view on available resources to local applications and—via a web browser extension—to web applications. The GPS sharing demo is able to run a location-based web-application on two devices; one which possesses a GPS receiver and one which does not. Nevertheless, the web application exploits the current GPS position on both devices.

Figure 5.10 sketches the architecture of the GPS sharing demo. Resources are either locally available or shared by other devices, in which case they are used via the *remote connectivity* component. It discovers other devices using Bluetooth SDP, coordinates interaction between devices, and provides resource data. Both local and remote resources are listed and made available in the *shared context model*. The shared context model plays the role of the Dynamic Data part of the Integrated Resource and Context Repository in our platform architecture, as described in Section 2.2 and depicted in Figure 2.2.

The GPS sharing demo is entirely event-based. Interaction begins, when devices discover each other via Bluetooth SDP. This triggers the remote connectivity to query the resource list of the shared context model on each device. The devices

compare their resource lists, and add metadata to their shared context models about new resources they did not list before. The shared context model of each device broadcasts a signal to all applications on the device in order to inform them about the newly available resources. Applications can subscribe to resources, so that the shared context model notifies them every time the value of a resource changes. If an application subscribes to a local resource, the shared context model invokes the respective adaptor to provide the resource value. If the subscribed resource resides on a remote device, then the shared context model invokes the remote connectivity, which subscribes to changes of the resource value via Bluetooth. Sharing may happen recursively, i. e., a remote subscription may yet again trigger another remote subscription, if a device shared a resource that it borrowed from yet another device.

For web applications the browser behaves like an ordinary application to the shared context model. It receives the signals about newly available resources and makes them available in a browser context model. The GPS sharing demo uses Telar DCCI [Brodt, 2007b], our implementation of the W3C Delivery Context: Client Interfaces (DCCI) [Waters et al., 2007], to make context data available to web applications. Web applications can subscribe to DCCI in order to be informed about newly available resources (DCCI properties). They may subscribe to resources to receive the resource values. In this case the web browser subscribes to the shared context model and propagates the value changes of the resource to DCCI and thus to the web application.

For robustness towards devices disappearing from the ad-hoc smart space, the GPS sharing demo uses a stateless communication protocol. In addition, it immediately removes all knowledge about a device and its shared resources and cancels all subscriptions if a communication error occurs with the device. This may trigger recursive removal of resources which were forwarded to other devices. If the device is just temporarily unavailable, it will be discovered again by Bluetooth SDP.

The GPS sharing demo uses a key-value-based data model. Resources are identified by a key consisting of a namespace and a local name, which is utterly similar to a URI. The value of a resource is a byte string. Thus, the type of the value must be known to an application. This data model is less powerful and simpler than RDF. Yet, it can be easily converted to an RDF-based model.

5.5.2 Spontaneous Team Meeting Solution (STEAMS)

STEAMS addresses the scenario of a meeting in which representatives of different companies participate. At the end of the meeting, they need a time slot for a follow-up meeting of a certain duration in which everybody has time. All participants possess a mobile device with an electronic calendar. Naturally, the calendars are backed by the

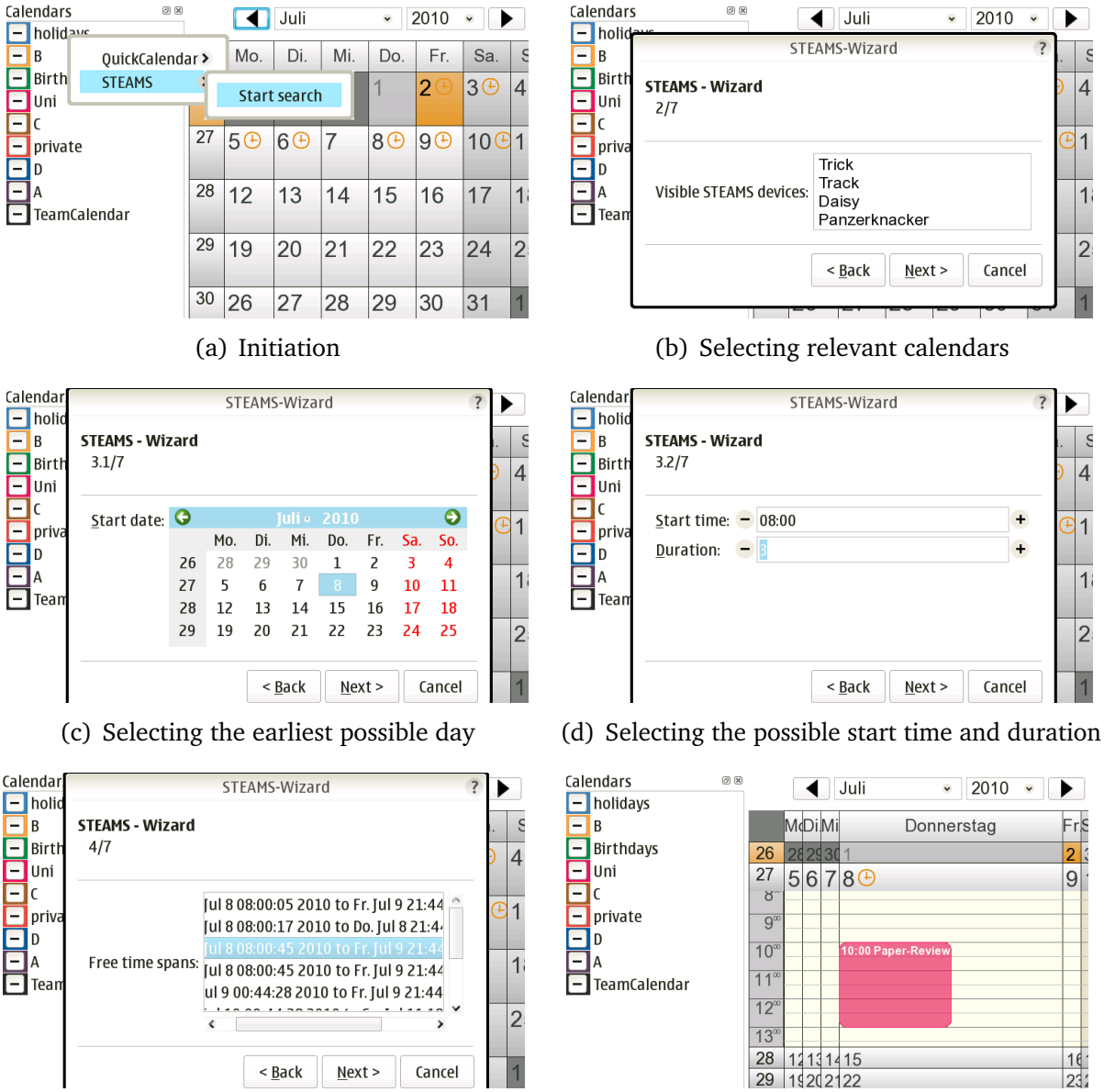
Figure 5.11: Spontaneous Team Meeting Solution (STEAMS) running on Nokia N800 and N810 Internet Tablets (adapted from [Reitschuster, 2010])

calendar server of the different companies (e. g. Microsoft Exchange). The calendars know exactly, when everybody has time for a meeting.

If all participants used the same calendar server, the server could intersect all calendars in question. However, such critical databases as business calendars (which are not infrequently integrated with the company e-mail system) are typically well protected and inaccessible for company-external clients. Nevertheless, the mobile devices in the meeting together possess all that is required to intersect the respective calendars:

- They know when the participants have free time slots.
- They possess wireless communication via Bluetooth.
- They are physically co-located, so that all devices can be reached via a Bluetooth scatternet.

STEAMS solves the problem without requiring access to calendar servers. As shown in Figures 5.11 and 5.12, STEAMS connects the devices of all meeting participants, requests their free calendar slots, intersects them, and proposes possible dates for a follow-up meeting. STEAMS provides a step-by-step wizard, that guides the initiating user through the process. Apart from allowing their calendars to participate in STEAMS, all meeting participants other than the initiator do not need to do anything. The steps are as follows:



(a) Initiation

(b) Selecting relevant calendars

(c) Selecting the earliest possible day

(d) Selecting the possible start time and duration

(e) Picking a possible date

(f) Accepted and scheduled meeting

Figure 5.12: Screenshots of the STEAMS wizard (adapted from [Reitschuster, 2010])

1. One participant of the meeting (*"the initiator"*) must initiate the search (Figure 5.12(a)). This triggers a resource discovery activity that looks for shared calendar resources (which obviously have to be discoverable).

2. STEAMS presents the found calendars and the initiator must manually select the relevant calendars (Figure 5.12(b)). This manual step ensures that nearby devices that do not belong to the meeting (e. g. in the meeting room next door)

do neither discard possible time slots nor receive possibly confidential meeting dates and descriptions.

3. When all participating calendars (and thus devices) have been determined, STEAMS asks the initiator to specify the time frame for the meeting (Figure 5.12(b) and 5.12(d)) as earliest possible start, duration, and latest end date and time.

4. STEAMS queries the devices for possible dates, intersects them, and presents them to the initiator (Figure 5.12(e)). The initiator selects one of them, presumably after proposing the date to the other meeting participants.

5. When the initiator has chosen the final date and time, STEAMS automatically inserts the follow-up meeting as a new event in all participating calendars (Figure 5.12(f)).

STEAMS uses flooding-based information propagation both for resource discovery and interaction. Request flooding, as described in Section 5.4.1 is used to discover calendar resources. Our evaluation showed, that a max. hop counter of two was sufficient—the devices are not expected to be more than 20 m apart from each other [Reitschuster, 2010]. The devices remember the routing paths via which other calendars were reached. All further communication attempts to route messages along these paths, if this is possible. As a fallback strategy to cope with the dynamic behaviour of ad-hoc smart spaces described in Section 5.1.2, the messages are flooded if the respective route does not work any more.

As a security precaution, the devices which the initiator did not mark as relevant for the meeting in step 2 (Figure 5.12(b)) are removed from the routing structures, so that they never receive any messages containing possibly sensitive calendar data. Also, the participating devices minimize the information they disclose. Instead of sending their busy times (which is what is really stored in their calendars), they only send possible time slots for the meeting.

STEAMS uses RDF for its data model, which includes information about people, their devices, their calendars, and calendar entries. All resource requests are translated to SPARQL queries to a local triple store.

5.6 Summary and Outlook

To put interoperability across devices into practice, we proposed the concept of *ad-hoc smart spaces,* which Sailesh Sathish from Nokia Research Center developed with

us. Ad-hoc smart spaces differ from infrastructure-based smart environments, as they concentrate on end-user devices and enable them to share their resources and context data with others near them spontaneously. They are autonomous, highly dynamic, complementary to other technologies, and designed for practical consumer-oriented use. We discussed user incentives for ad-hoc smart spaces and presented a middleware architecture as a further refinement of the system architecture of Chapter 2.

The highly dynamic network structure of ad-hoc smart spaces challenges resource discovery. We examined self-organizing decentralized resource discovery protocols for Bluetooth-based ad-hoc smart spaces with different strategies towards information replication, scatternet routing and information propagation. Our simulations show that Request Flooding performs best for small network sizes (≤ 5 devices), in spite of its exponential growth. In medium to large networks dedicated routing structures and an increased degree of replication pays off. The Random Replication protocol wins here, as its randomness makes it robust to changes in the network.

Finally, we presented two sample applications for ad-hoc smart spaces which we implemented on Nokia Internet Tablets. They follow different interaction paradigms: while the GPS sharing demo is entirely event-based, STEAMS accesses resources in a get/put fashion. These applications allow non-technical users to understand the potential of the ad-hoc smart space concept.

Future Work

Ad-hoc smart spaces are still a relatively new topic that is eagerly developed [Sathish, 2011]. With a broader uptake of the concept, new use cases and requirements will emerge. What concerns our evaluation of resource discovery protocols, further work could be invested to determine the ideal degree of replication r of the Random Replication protocol. Also, different routing algorithms could be investigated. Finally, simple static context data, e. g., *weather="sunny"*, could be included in resource advertisements directly, omitting the need to ask for it explicitly. In the long term it may be worth developing new wireless communication technologies that are better suited for this kind of ad-hoc interaction.

The GPS sharing demo already illustrated a use case for interaction with web applications. The following chapter addresses this topic in further detail.

6

Interoperability with Web Applications

Chapters 3 and 4 addressed the data management layer of our architecture and Chapter 5 added interoperability across co-located devices. What remains from our requirements listed in Section 1.2 is interoperability with web applications. With mobile devices becoming more and more powerful and mobile web browsers reaching desktop class, web applications are used on mobile devices to an increasing degree. Users describe their activities in microblogging sites, keep track of friends and business contacts on social networks, and publish photos and videos on media sharing websites. Thus, they manage large amounts of personal data via web applications. This must be taken into consideration in mobile data management scenarios.

There are two directions for interoperability with web applications. On the one hand, this data is a rich source of context information, and local applications on the mobile device can benefit from it. For example, the address book could be made more complete by synchronizing it with the social networks in which the user is active. If the social network contacts disclose their birthday, this information could as well improve the calendar or a dedicated birthday-alarm widget. Messages from mircroblogging sites, social networks, chatrooms, and web forums could be integrated into a general messaging client, that also includes SMS text messages and e-mails. Also, as users are frequently active in more than one social network, media sharing site, or cloud storage service, the mobile device could be a reasonable center point that keeps a catalogue of metadata on what the user published where. Many other use cases are possible.

On the other hand, web applications can benefit from context data that is locally available on the device. This includes application data, such as the address book, the calendar, or text messages. Also, web applications can benefit from data obtained from sensors of the device. This includes, must notably, GPS and cameras, but also accelerometers, or even wireless communication subsystems. The latter may, e. g., provide information about co-located devices and thus people nearby, which may be very interesting for a social network.

This chapter addresses interoperability with web applications. First we give an overview on the foundations of web application technology, its historical background, context provisioning, and interoperability in Section 6.1. We propose a web browser interface for interoperability of local and web applications in Section 6.2. In Section 6.3 we address a particular interoperability aspect of resource and context data with web applications: context-aware mashups. We present a generic web platform for context-aware mashups, which we demonstrated in [Brodt and Nicklas, 2008] and published in [Brodt et al., 2008]. In addition, we present a location-based mapping mashup system based on the Nexus platform, which we demonstrated in [Brodt and Stach, 2009]. In Section 6.4 we apply context-aware web application technology in the domain of mobile location-based browser games, as published in [Brodt and Sathish, 2009]. Section 6.5 concludes the chapter.

6.1 Foundations

This chapter requires a deep understanding of web application technology, so we start with a look into the rear mirror and recapitulate the historical development of the web and its technologies. We shortly introduce ongoing developments concerning client-sided storage and address context provisioning for web applications.

6.1.1 Background: From Static Documents to Interactive Web Applications

When Tim Berners-Lee developed the World Wide Web (WWW) the original goal was to create a system that facilitates exchanging scientific documents among researchers [Berners-Lee, 1989, 1991]. A central concept was to create hyperlinks that allow easy navigation between the documents, thus creating a *web*:

> *“The WorldWideWeb (W3) is a wide-area hypermedia information retrieval initiative aiming to give universal access to a large universe of documents.”*
>
> — Tim Berners-Lee [1992]

Thus, the WWW was initially designed to browse static *documents* rather than to interact with *applications*. The web browser retrieves a document from a web server via the Hypertext Transfer Protocol (HTTP). It evaluates the Hypertext Markup Language (HTML) code, in which the document is described, and renders it on the screen. When the user activates a hyperlink to another document, this document is retrieved, possibly from a different web server, and displayed, thus replacing the document that contained the link. The communication between the web browser and the web server is entirely stateless, as every request contains all necessary information to complete it. Fielding [2000] later coined the term *Representational State Transfer (REST)* for this principle.

Server-Side Scripting

Soon, the web was extended to pages that were not served statically but created dynamically by the web server. This allows complex multi-page websites, often backed by an RDBMS. Such *server-side scripting* is a key enabler for web *applications*, as it can be used to process user input that is sent to the web server via HTTP. In return, the web server creates an HTML document containing an updated presentation. To express it in terms of the Model-View-Controller (MVC) pattern [Reenskaug, 1979], the database contains the model, i. e. the state of the web application, the generated HTML pages constitute the view, and the server scripts provide the controller that processes user input, updates the model and generates the view.

Cookies

Unlike for static documents, for interactive applications it is desirable to keep a state for each session and possibly for each user; e. g., to manage the shopping cart of an online store or user account information. For this purpose *cookies* were introduced [Kristol and Montulli, 1997]. To keep HTTP stateless, they keep all session state on the client side. Cookies are key-value pairs that allow web applications to store up to 80 kB of data inside the web browser (20 cookies of 4 kB each). A cookie is created when the web server includes it in an HTTP reply. Subsequently, the browser includes it in all HTTP requests to the particular website. The web server can remain stateless, as mandated by REST, since every request contains all necessary information.

Client-Side Scripting: JavaScript and Ajax

In 1995 a pre-release of Netscape Navigator introduced a script language which was first published as *LiveScript* and later renamed to *JavaScript*. It could be embedded

into HTML code and was, among other things, capable of checking user inputs to HTML forms on the client side before sending them to the server. Using such *client-side scripting*, small pieces of application logic could be executed on the client side. This saved a few long-lasting server roundtrips on illegal user-input. Yet, the actual controller of a web application remained on the server side.

In 1997 Netscape Navigator made another important innovation: the Document Object Model (DOM). DOM provides an interface for client-side scripts to an object-oriented representation of the HTML page that is currently loaded. Using DOM, client-side scripts may modify the current page.

The XMLHttpRequest object, which Microsoft introduced in 1999 [Hopmann, 2007], finally added a clean solution for client-sided scripts to create HTTP requests and receive data from the web server in the background. Before, dirty hacks had to be used that misused many HTML elements, such as hidden frames, invisible forms, or dynamically created `<script src="...">` elements, which were never designed for that purpose. Despite its name, an XMLHttpRequest is not at all restricted to XML data but may retrieve arbitrary content. Finally, the combination of JavaScript, DOM, and the XMLHttpRequest enabled powerful and responsive web applications on the client side. For these technologies (in addition to HTML and CSS) the umbrella term *Asynchronous JavaScript and XML (Ajax)* was coined.

Ajax changed the anatomy of a web application fundamentally. It allows processing user input on the client side and updating the view or user interface of a web application through DOM, without sending a request to the web server. Thus, in terms of the MVC pattern, the controller of a web application resides partly on the web server and partly on the client. Requests to the web server, which take seconds even on fast connections, are only required to fetch additional content. This causes usability of Ajax applications to be almost as good as usability of installed desktop applications, which was impressively demonstrated, e. g., by Google Docs providing an entire office suite in the web browser. The only aspect that is totally ignored by Ajax technology is interoperability with client-sided data; this was simply not the focus when the web was initially developed.

6.1.2 Browser-local Storage

Web applications have to load all code and data they require from the web server every time they are used. Gears [Google Inc., 2007] first extended the web browser by local storage, thus providing a client-sided cache. Before, the only way to keep a client-sided state was to use cookies. And due to their size limitation, cookies are typically not used to store actual data, but only keep a key to data on the server side, e. g. the ID of the user account or shopping cart. Using browser-local storage, data

such as the e-mails in a webmail application does not need to be loaded every time. It is sufficient to synchronize changes with the web server. Gears includes further components that allow caching the entire client of a web application thus enabling offline use. As stated in Section 2.1.3, several proposals were made in the context up the upcoming HTML5 standard [Mehta et al., 2010; Hickson, 2010, 2011] are currently in development with the goal of a standardized interface for browser-local storage; the latter being closest to standardization. All of them have in common, that they strictly separate the data of different web applications, thus preventing interoperability.

6.1.3 Context Provisioning for Web Applications

Various technologies were developed to enable web applications adapting to the *context* of the client, particularly context data at the client side. At this, one must differ between static and dynamic context provisioning. Finally, the *Delivery Context: Client Interfaces (DCCI)* and domain-specific web APIs are specific examples for dynamic context provisioning.

Static Context Provisioning

For static HTML pages, the web browser may include one or more *HTTP headers* in the HTTP requests. This is used, e. g., to inform the web server about the particular web browser in use, its understood image formats, etc. The web server may use these values to return a version of the web site which the browser supports.

In the Wireless Application Protocol (WAP), the mobile browser accesses a web page via a WAP gateway, which translates it into a version tailored to the capabilities of the mobile device. HTTP headers, which are included in every request, become too large when listing all device capabilities. Thus, the User Agent Profile (UAProf) was introduced. It only includes one HTTP header, containing a link to the profile of the particular device. The WAP gateway may retrieve it once and cache it for further requests. Thus, UAProf turns client-sided context data into data on the server side, which is only possible for static data.

Dynamic Context Provisioning

HTTP headers and UAProf are intended for static context data, e. g. browser version, screen resolution, or supported media formats. They do not well support the dynamic context of a mobile device, such as the device neighborhood, available network connections, screen orientation, or, most notably, the user's location. In [Brodt,

2007a], we implemented a prototype that sent the current location as HTTP headers with every request. The web server could return a static page for this location. But the page could not react dynamically when the user moved. For context-aware web applications, a context provisioning mechanism is required that enables dynamic reaction on context changes in the (Ajax) client of a web page. In [Brodt et al., 2008] we stated the following requirements for context provisioning:

Asynchronous Notifications. The web application needs to be notified of changes in the user's context in order to react accordingly.

Mutability. As resources may become inactive or new resources may appear, the context model needs to be capable of reflecting these changes.

Search. The web application needs a means to find out, which resources are available. This in turns requires *metadata* describing the resources.

Control. In order to utilize a sensor, the web application needs some degree of control over the sensor, e. g., to activate it or to trigger a measurement.

Standardization. It is important to standardize interfaces when exposing an API to a huge multi-platform and multi-vendor system such as the web.

Privacy. The user must be in control of what information is disclosed about her current situation. The context provisioning mechanism must ensure privacy before the context data reaches the web applications.

The W3C Delivery Context: Client Interfaces (DCCI)

The W3C Delivery Context: Client Interfaces (DCCI) [Waters et al., 2007] specify a generic context-provisioning mechanism for web applications, which suits these requirements well. As already introduced in Section 2.1.3, DCCI exposes a number of *properties*, which are identified by namespace and name, and carry a value. DCCI does, however, not further define the particular properties; they must be defined and standardized separately. The properties are organized hierarchically: the `DCCIProperty` interface inherits from DOM `Element`, such that the DCCI tree is also a DOM tree. DCCI uses the DOM event model [Pixley, 2000] to provide asynchronous notifications. A web application can register for events, such as the change of a property value or the removal of a property, and is notified via the provided JavaScript event listener function. In addition to that, it is possible to add and remove properties dynamically, thus achieving mutability. Moreover, DCCI properties can have a metadata interface and the DCCI tree is searchable. DCCI

does not directly support control over local sensors. The metadata interface, which is not further specified, could be used for this. Also, for simple notions of control, such as activating or triggering a sensor, the event model can be used. Assuming that the sensor value is only needed when at least one event listener is registered to the respective DCCI property, the property can (de)activate or trigger the sensor accordingly. It was attempted to standardize DCCI at W3C, but it only reached the official status of a *W3C Candidate Recommendation*. DCCI does not directly specify privacy or access control, but leaves it to the implementation. It does, however, not obstruct privacy, either.

As a part of our work on the TELAR Mashup Platform, which we present in Section 6.3.1, we created TELAR DCCI, the first full implementation of DCCI, which is available as open source [Brodt, 2007b].

In 2010 DCCI was discontinued as part of the closure the Ubiquitous Web Applications Working Group.

Domain-specific Web APIs

While the Ubiquitous Web Applications Working Group was closed at W3C, the Device APIs Working Group [Berjon et al., 2010] developed client-side APIs that enable web applications that interact with devices *services* such as Calendar, Contacts, Camera, etc, as discussed in Section 2.1.3. Also, the W3C Geolocation Working Group works on a specific interface for client-side location information provisioning [Popescu, 2009]. I. e., instead of one generic context provisioning mechanism like DCCI, a specific API for every domain is specified.

6.1.4 Summary

The web has come a long way from distributing static documents to a complex distributed application framework based on many different technologies including server-side scripting, Ajax, browser-local storage, and even context provisioning. What is missing is interoperability of web applications with resource and context data that is available on the client side. This was not a problem in the static document-oriented web. Yet, dynamic web applications, which we use on a daily basis, still have not caught up with this aspect. For this purpose we define a web browser interface for interoperability with local data in the next section.

6.2 Achieving Local Interoperability: The Repository Web-API

There are many ways to achieve (aspects of) interoperability with web applications, as described in the introduction of this chapter. The most prominent ones today are synchronization with cloud services, service APIs, and web browser extensions. Most mobile software platforms today come strongly coupled with a particular cloud service, which is usually provided by the vendor of the software platform. Examples include Google's Android platform, which is deeply integrated with Google services, or Symbian OS and maemo and Nokia's Ovi service offerings. All these services provide server-sided cloud storage, which is basically nothing but the database part of server-side scripting explained in Section 6.1. Content data can be synchronized with these services and they often offer interfaces to share data with other services, given the users authorization, e. g., via OAuth [Hammer-Lahav, 2010]. This way, arbitrary web applications may utilize personal data of the user; first, the data is synchronized with cloud storage, then the server-sided part of the web application retrieves it from there. Similarly, many web applications, including social networks, shopping sites, or media sharing services, offer proprietary server-sided service APIs. Many mobile applications obtain data from there. However, only web browser extensions allow immediate access of context data that is intrinsically available on the client side only. Most notably, data from local sensors is only available on the client side. As discussed in Section 6.1.3, if immediate reactions on context changes are needed, they must take place on the mobile device, as only client-side scripting achieves truly low latency. In addition to that, cloud services and service APIs require constant network access, which is despite all progress far from a given.[1] Thus, it is reasonable to enable interoperability with web applications on the client side as well.

For stream-based data, such as sensor values, approaches like the event concept of DCCI are suitable, for exercising control domain-specific web APIs are appropriate, and for client-sided cache of server-based resource data browser-local storage is a good solution. What is missing is a client-sided method to manage resource and context data in a way that makes the data accessible for other applications, as discussed in Chapter 2. For this purpose we developed the Repository Web-API, which puts the web application interface of our architecture of Figure 2.2 into practice. We borrow the event listener concept from DCCI (and others), the control concept from domain-specific web APIs, and the concept of queries to a database that resides locally on the client from browser-local storage and cookies. Yet powered by the Integrated

[1] As an ironic side note: In a meeting at the Google office in Munich in 2011 the Google staff strongly emphasized the advantages of cloud storage making client-sided data management unnecessary. On the way home we lost all network connectivity as soon as the train had left Munich and entered more rural areas.

Resource and Context Repository (and ultimately RDF) we add interoperability accross applications and domains and between local and web applications.

6.2.1 API Definition

Listing 6.1 gives the formal definition of the Repository Web-API in Web IDL notation [McCormack, 2011]. The Integrated Resource and Context Repository is represented by a global object implementing the Repository interface. It offers only the `open()` call, which returns a handle to access the repository with a set of access roles. The handle allows executing queries and updates on the repository, as well as listening to changes in the repository. Furthermore, it provides access to controller interfaces for particular resources that offer further functionality, e. g. to trigger a sensor reading or activate a device. The controller interface depends strongly on the resource in question and the particular implementation is beyond the scope of this work.

A SPARQL query or update that is passed to a repository handle is executed with the access roles of the handle. An update returns the number of modified triples. A query returns an object implementing the ResultSet interface. It can be used as an iterator to step through the returned SPARQL variable bindings. The particular values can be accessed by their index (starting from 0), or by the `get()` call, which returns a JavaScript object representing a result row. This JavaScript object has a property for each result variable and thus integrates well with the language concepts of JavaScript. In case of an error or invalid usage, the Repository Web-API throws a `RepositoryException` with the respective error code.

Applications that need to react on changes of the device context prefer to observe resources that are provided by local sensors rather than querying them. Using the repository handle they can register an event listener that is notified when the resource of the given URI creates the desired event. In this case the event listener is called with the given event type and a result set that contains the resource and its attributes.

Listing 6.2 illustrates how the Repository Web-API is used to query the Integrated Resource and Context Repository. First, the repository is opened with a set of access roles (note that an access role is really identified by a URI). A repository handle is returned and the web application checks whether it was granted the required roles. If so, it executes a SPARQL query and iterates through the returned results. Otherwise, the web application should continue without repository access; in the example it simply emits a message to the user.

```
interface Repository {
  RepositoryHandle open(DOMString[] requiredRoles);
};

interface RepositoryHandle {
  readonly attribute DOMString[] roles;
  boolean hasRole(DOMString role);

  ResultSet executeQuery(DOMString sparqlQuery);
  unsigned long executeUpdate(DOMString sparqlUpdate);

  void addEventListener(DOMString resourceUri, DOMString type,
    EventListener listener);
  void removeEventListener(DOMString resourceUri, DOMString type,
    EventListener listener);

  any getController(DOMString resourceUri);
};

interface ResultSet {
  boolean isValidRow();
  void next();
  void close();

  readonly attribute unsigned long fieldCount();
  readonly attribute DOMString[] fieldNames();
  getter DOMString fieldName(unsigned long fieldIndex);
  getter any field(unsigned long fieldIndex);
  getter DOMString fieldTypeUri(unsigned long fieldIndex);
  getter DOMString fieldTypeUriByName(DOMString fieldName);
  any get();
};

exception RepositoryException {
  const unsigned short UNSPECIFIED_ERROR = 0;
  const unsigned short ACCESS_DENIED = 1;
  const unsigned short INVALID_RESULT_ROW = 2;
  const unsigned short INVALID_FIELD_INDEX = 3;
  const unsigned short INVALID_FIELD_NAME = 4;
  const unsigned short INVALID_RESOURCE_URI = 5;

  readonly attribute unsigned short errorCode;
};
```

Listing 6.1: Web IDL [McCormack, 2011] Definition of the Repository Web-API

```
// access repository to read the calendar. May trigger access role dialog.
var repositoryHandle = repository.open(["ReadCalendarRole"]);
if (repositoryHandle.hasRole("ReadCalendarRole")) {
  var resultSet = repositoryHandle.executeQuery(
      "SELECT ? title ? startTime ?endTime WHERE { ... }"
  );
  for (; resultSet.isValidRow(); resultSet.next()) {

    // access result field by their index:
    var stringForFurtherUse =
        resultSet.field(0) + ": "
      + resultSet.field(1) + " .. "
      + resultSet.field(2);
    ...

    // alternatively access results using the get() call:
    var row = resultSet.get();
    var stringForFurtherUse =
      row.title + ": " + row.startTime + " .. " + row.endTime;
    ...

  }
  resultSet.close();
} else {
  alert("No permission to read calendar entries.");
}
```

Listing 6.2: Sample JavaScript code using the Repository Web-API to obtain calendar data from the Integrated Resource and Context Repository (API calls)

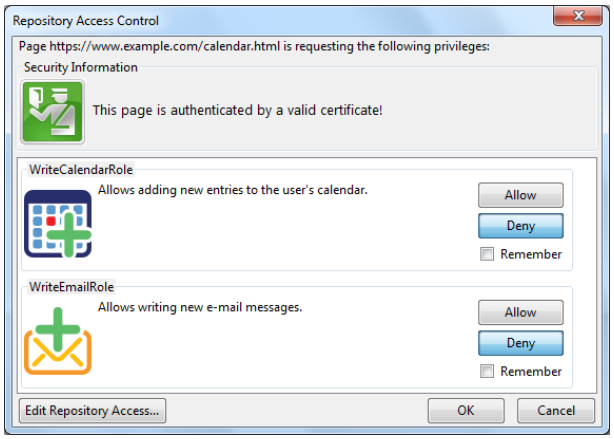
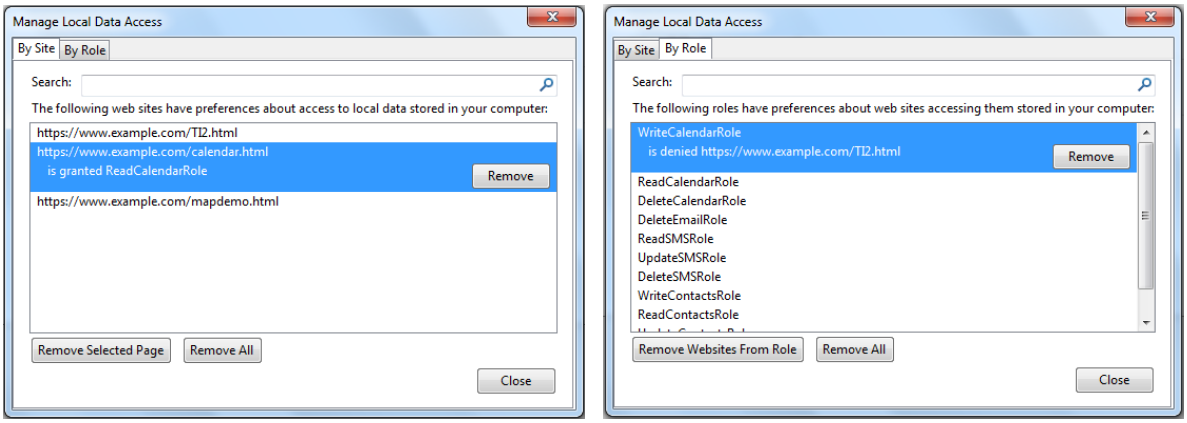


Figure 6.1: The access role dialog of the Repository Web-API



(a) Manage access roles by web site (b) Manage web sites by access role

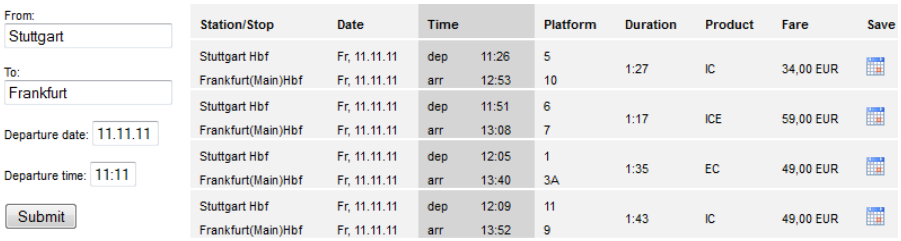
Figure 6.2: The access role editor of the Repository Web-API

6.2.2 Access Control

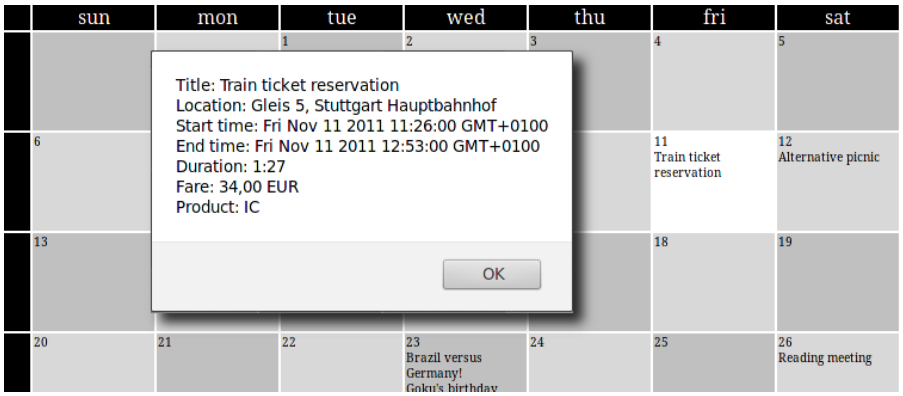
As stated in Section 2.5, it is very important to have powerful access control mechanisms in place when exposing the Integrated Resource and Context Repository to web applications. For this purpose, the Repository Web-API is explicitly designed on top of the access role concept presented in Section 2.5: The Integrated Resource and Context Repository can only be accessed through a repository handle and the handle is inherently bound to a set of access roles. The access roles of a handle are the intersection of the roles for which a web application asked when it opened the repository on the one hand, and of the roles which the user assigned to the web application on the other hand. When a web application opens the repository, the Repository Web-API checks whether the user has granted or denied the asked access roles persistently. If so, the Repository Web-API automatically grants or denies the respective roles. If one or more roles were not persistently decided for the web application, the Repository Web-API shows an access role dialog for the user to decide on the access roles, as shown in Figure 6.1. In order to ensure the authenticity of the web application, it should use HTTPS and a valid certificate. The access role dialog warns the user if it could not authenticate a web application. The access roles which were granted or denied to web applications persistently may be changed any time using the access role editor shown in Figure 6.2.

6.2.3 Sample Web Applications

To demonstrate the Repository Web-API we implemented it as an extension of the Firefox web browser carried out as a number of XPCOM components written in

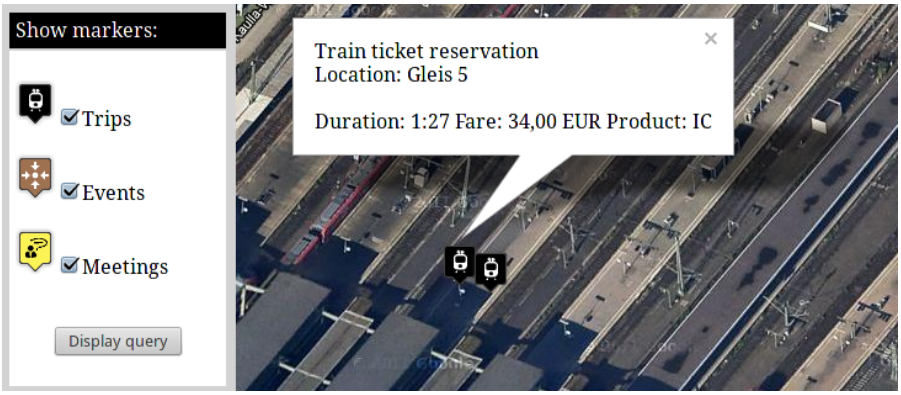


(a) Travel application allowing to store train reservations in the calendar



(b) Calendar application showing a train reservation

(c) Map showing resources related to travelling, events and meetings



(d) Map application showing the details of a train reservation

Figure 6.3: Sample applications to demonstrate the Repository Web-API

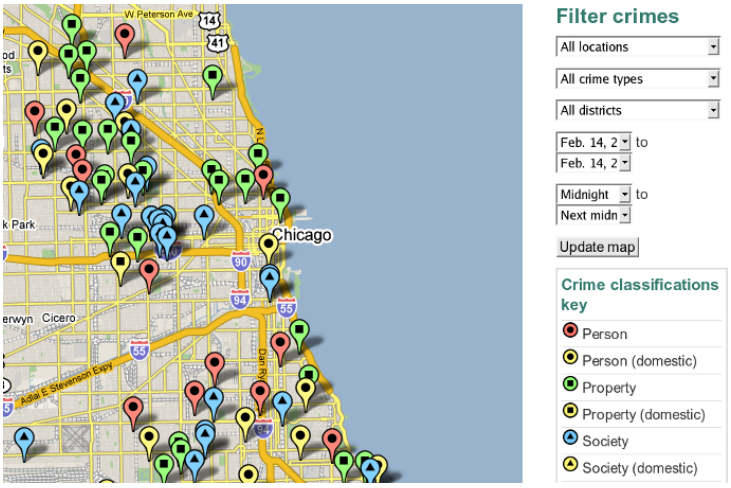


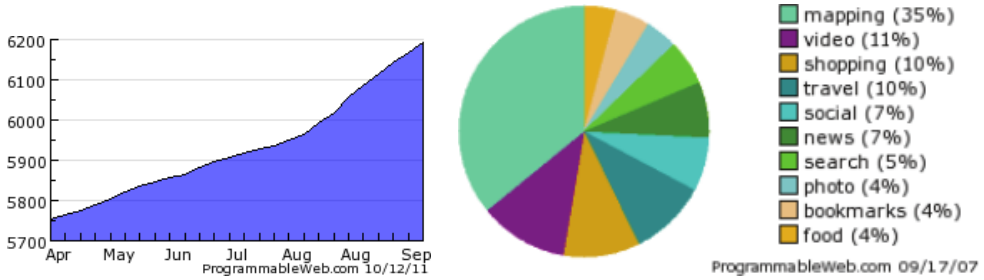
Figure 6.4: Chicago Crime Mashup visualizing crime statistics on a map (Screenshot of `http://www.chicagocrime.org`, retrieved 2007)

JavaScript. Moreover, we wrote three sample web applications that share resource data via the Integrated Resource and Context Repository. First, a travel application searches for train connections and allows the user to store a particular connection in the repository. Second, a calendar application manages meetings, but is capable of listing the train connections as well. Finally, a map application displays resources carrying geographic references. Without further knowledge about their particular types, it is capable of extracting the spatial coordinates, their title, description, and type attribute, which is enough to place a corresponding icon on a map.

Figure 6.3 shows screenshots of the three web applications: A train ticket reservation made with travel application and stored in the Integrated Resource and Context Repository via the Repository Web-API (Figure 6.3(a)) appears in the calendar application (Figure 6.3(b)) as well as in the map application (Figure 6.3(c) and 6.3(d)).

6.3 Local and Remote Interoperability: Context-aware Mashups

As stated in Section 6.1.1, Ajax separates the presentation of web pages from the actual content. Ajax web pages can download HTML fragments to replace parts of the page, or they can retrieve raw data from the server, evaluate it, and create an appropriate presentation for it. Especially in the latter case, the web server does not at all serve documents any more; it becomes a web service which exposes some sort of stateless remote procedure calls which can be viewed as a service API. These APIs are intended and designed for the Ajax client of the website. Yet, many web sites extend them to documented public service APIs for general use. As a consequence,



(a) Listed mashups in 2011 (b) Distribution over mashup genres in 2007

Figure 6.5: Mashup statistics (Screenshots of `www.programmableWeb.com`)

the raw data of a website is openly accessible, which makes it possible to utilize it for a different purpose. O'Reilly [2005] identified this as one of the key principles of the so-called *web 2.0*.

The public service APIs layed the grounds for web *mashups*: web application hybrids that integrate data from different sources to provide a value added service, for instance, enriching search results from one source (e. g. a hotel finder) with information from others (e. g. recommendations and pictures). Typically, mashups are created dynamically from existing data sources that have no knowledge about their participation. When Google published the JavaScript API for Google Maps (soon followed by competitors), a special category of mashups emerged, to which we refer as *mapping-mashups*. Mapping-mashups display information from different sources on a map, mostly using the Point of Interest (POI) metaphor. One of the first popular mashups was `chicagocrime.org`, which visualized crime statistics from the Chicago Police Department on a map, as shown in Figure 6.4. It created the additional value of a graphical suggestion for safe and less safe ares, which neither the raw police reports, nor the bare map contained.

The Mashup portal `programmableWeb.com` listed 2331 Mashups in September 2007, with an average of 3.19 new mashups per day, in October 2011 the number of listed mashups reached nearly 6200 (see Figure 6.5(a)). Figure 6.5(b) shows the distribution over different genres: over one third are mapping-mashups, 15 % are multimedia mashups (video and photo). Floyd et al. [2007] show how mashup techniques can be used for rapid prototyping in user-centered software development processes. Wong and Hong [2007] showed that mashups can be even used for end-user programming. IBM emphasizes the great benefits of so-called *Enterprise Mashups* [Jhingran, 2006], information heavy applications that integrate distributed information within an enterprise in a quick and dynamic way.

Most of the existing mashups are programmed manually. However, a number of mashup platforms exist that facilitate the development: *Mash-o-matic* [Murthy et al., 2006] can be used to generate geo-mashups based on so-called superimposed information. The *Openkapow* platform [Kapow Technologies Inc.] realizes mashups as a combination of so-called robots, which extract information from RSS streams, web services, or via screen scraping. With online tools like *Yahoo! pipes*[Yahoo! Inc., 2007], mashups can be built out of predefined components and combined using interactive drag-and-drop interfaces. IBM's *QEDWiki* [IBM Inc., 2007] provided an Ajax interface to combine user interface components that are connected to external data providers. It finally lead to the IBM Mashup Center, an enterprise mashup platform, supporting rapid assembly of dynamic web applications [IBM Inc., 2009]. Intel's *MashMaker* [Ennals and Gay, 2007] allows creating of complex mashups by browsing, rather than writing code, applying the principles of functional programming.

6.3.1 The TELAR Mashup Platform

In [Brodt et al., 2008] we presented a generic web platform for context-aware mashups, which we also demonstrated at the EDBT'08 conference [Brodt and Nicklas, 2008]: the TELAR Mashup Platform. It combines mashups and mobile context-aware applications. By integrating multiple data sources into one presentation, new services can be created that are tailored to the user's personal needs. And by using local sensor data on a mobile device, this presentation can be adapted to the user's current situation. The TELAR Mashup Platform was designed for the following requirements:

- Adaptation to the user's context should be based on sensors which are built into the mobile device or locally connected. Although the main focus is on location data gained from a GPS receiver, the solution should allow arbitrary local sensors.

- The mashups should be user-centric, i. e. the user of a mobile device should benefit from the mashups, rather than a remote person or service provider.

- The mashups should be viewable with the web browser of the mobile device. This prevents a native solution on the mobile device and has potential influence on performance.

- There should be a non-adaptive version of mashups in case no context information is available. This allows viewing the mashups on sensor-equipped mobile devices as well as on desktop computers.

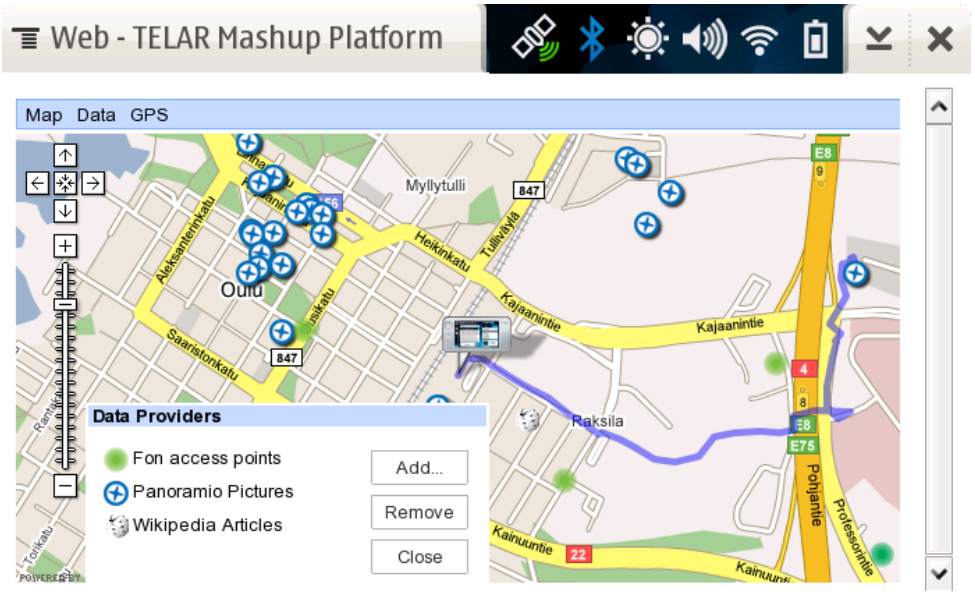


Figure 6.6: Screenshot of the Telar Mashup Platform on a Nokia N810

- Multiple data sources using arbitrary data formats and interfaces should be integrated into the mashups. The user should be able to add and remove data sources at runtime, according to her current interest.

Figure 6.6 shows a screenshot of the TELAR Mashup Platform. It creates a mapping-mashup which incorporates the user's current position, the position history, and Points of Interest (POIs) from different third-party data providers. All data is integrated in a map-based presentation.

Architectural Overview

Figure 6.7 shows the system architecture of the TELAR Mashup Platform. As with a typical Ajax-based mashup, there are three tiers: A mashup is viewed in the client tier. The web browser loads the mashup page and starts the JavaScript code of the mashup client Ajax application. The mashup page is loaded from the mashup server, which resides on the Internet and constitutes the server tier. Data offered by third-party data providers is used, which are distributed throughout the Internet. The map is loaded from a map service, which, together with the data providers, makes up the data provider tier. Note that the data provider tier is outside of the organizational boundaries of the mashup.

A mashup consists of an HTML page which imports the JavaScript files of the mashup client. It needs to be configured (most notably, the data providers to use

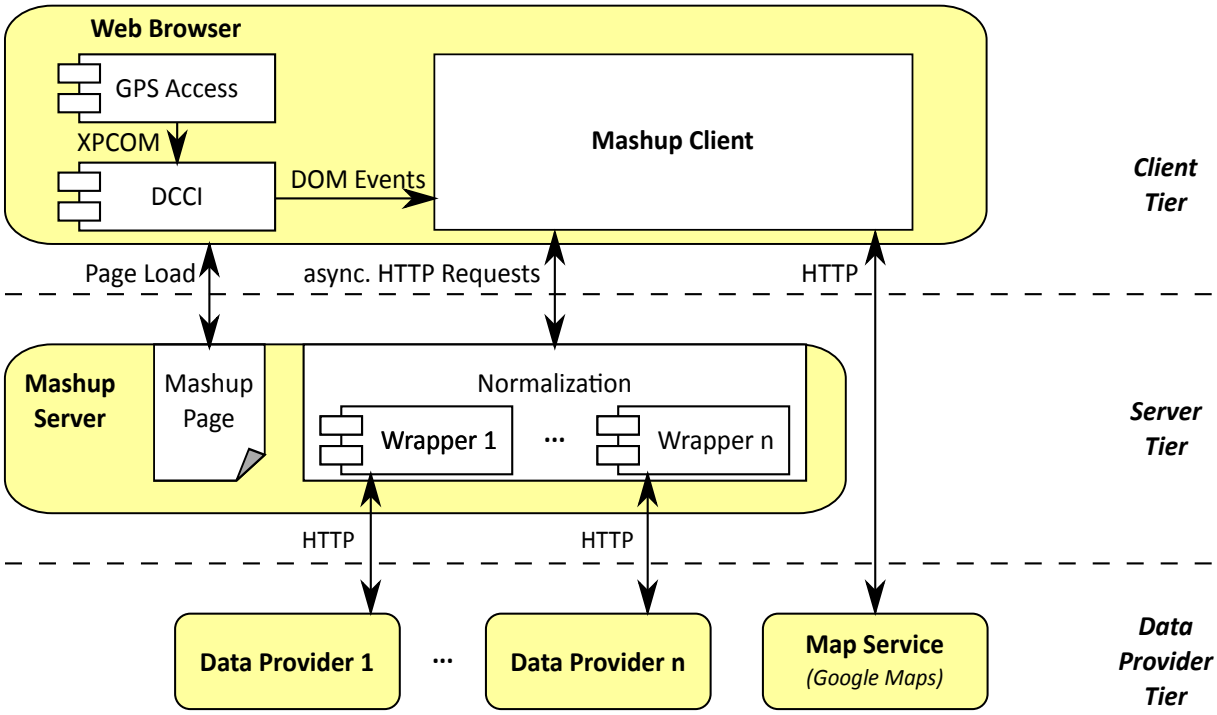


Figure 6.7: The system architecture of the TELAR Mashup Platform

and the initial position of the map), but not programmed. When the mashup page is loaded, the web browser retrieves and instantiates the mashup client which then reads the configuration asynchronously. The mashup client then constructs the user interface. It displays a map and visualizes the transformed POIs data from the data providers. In order to cope with the heterogeneity of data formats and interfaces used by the different data providers, a normalization layer is required.

Our solution to integrate the data from various different data providers is a simple wrapper approach. Small and independent wrapper scripts impose an abstraction layer on the data providers creating a consistent REST interface to retrieve the data. The wrappers query the data providers and convert the data into a single well-known format. As our scenario focuses on POIs, a common data model is easy to find. The fact that the wrappers need to be programmed makes user-programming difficult, on the other hand virtually any data source can be accessed. However, given a sufficient amount of wrappers, one can choose which data providers to include in a mashup. In addition, wrappers can be parametrized giving the user or the mashup creator some control. For other scenarios with less diversity, other, possibly automatic normalization approaches are applicable (e. g., data providers solely providing RSS feeds, as used by Yahoo! pipes).

Context information, such as the user's location, is integrated into the mashup via TELAR DCCI, our open-source implementation of the W3C Delivery Context: Client Interfaces (DCCI) [Brodt, 2007b]. It consists of two components: the DCCI module and the GPS access module. The DCCI module implements the DCCI specification [Waters et al., 2007] and constitutes the interface for providing context data to web pages. The mashup client registers a JavaScript event listener to the DCCI module and is notified via a DOM event about every position update. The GPS access module connects to the GPS receiver of the mobile device and ships the location information to the DCCI module. Dividing the context provisioning framework into a client interface and a provisioning module allows further context provisioning modules to be added later on. The modules are implemented as individual XPCOM (the component model of the Mozilla browser) components, so that further provisioning modules are easily possible.

The data flow works as follows: Whenever the GPS access module obtains a new location from the GPS device, the location information is updated in the DCCI module. The mashup client, which is registered as an event listener to the DCCI module, is notified about the change via DOM events. Subsequently, the mashup client updates the user's location on the map and centers the map to the new location, if it is in *following mode*. If the area shown on the map has significantly changed, the mashup client sends asynchronous HTTP requests to the wrappers, in order to obtain POI data for the new map area. The wrappers translate these requests into calls to the particular APIs of the data providers and convert the results into a unique data format understood by the mashup client. Finally, the mashup client reads the reply sent by the wrappers and visualizes the POI data on the map.

Mashup Development using the TELAR Mashup Platform

In order to create a context-aware mashup using the TELAR Mashup Platform, the following steps need to be taken:

1. Select the data providers. Create the respective wrappers, if not yet available.
2. Write an HTML page, into which the map presentation should be embedded.
3. Deploy the mashup client, the wrappers, and the HTML page to a web server.
4. Configure the mashup client defining the initial map area (center point and zoom level) and the data provider wrappers to use.

No Ajax programming is required, as the mashup client handles all map interaction, displays the POIs and integrates the user's location.

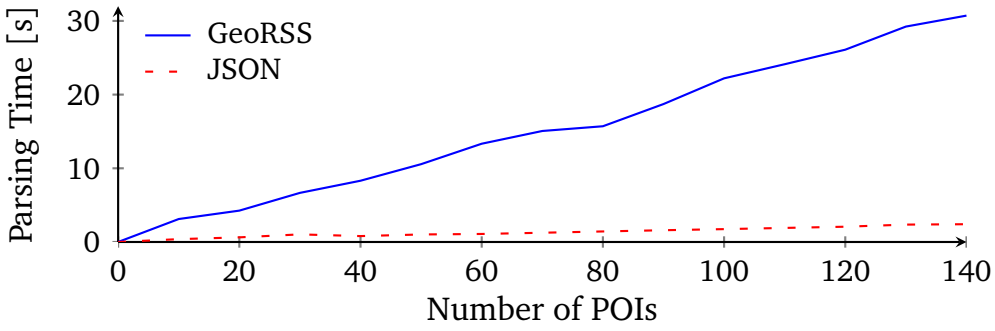


Figure 6.8: Time for processing Points of Interest (POIs) from GeoRSS and JSON data formats, measured on a Nokia N810

Performance Optimizations

When the mashup client and some wrappers were implemented, first tests showed that performance was insufficient. Working fine on a state-of-the-art desktop computer, it could take minutes until a mashup was completely constructed on a mobile device. We made a similar experience on an old Pentium II machine. A first analysis revealed that most of the time was spent for parsing the POI data which the mashup client retrieved from the wrappers, which is done by JavaScript code interpreted in the browser. In contrast to a desktop computer, the processor of the mobile device was simply not powerful enough to do this job quickly.

Our first version used an extended version of GeoRSS [Singh et al., 2009] as serialization format for the POIs. GeoRSS is a standardized, simple, and popular format based on RDF and XML, as discussed in Chapter 4. In order to improve performance, the standardized GeoRSS format was replaced by a proprietary format based on the JavaScript Object Notation (JSON) [Crockford, 2006]. As JSON is a subset of JavaScript, it can be parsed very efficiently using the `eval()` JavaScript function, which the web browser implements in native code. Also, JSON is significantly more concise than GeoRSS.

For both formats, GeoRSS and JSON, we measured the time for unmarshalling various amounts of POIs to JavaScript objects on a Nokia N810 Internet Tablet. Figure 6.8 shows the results: parsing POIs from GeoRSS took significantly longer than from JSON. Parsing 100 POIs of GeoRSS data took more than 20 seconds, whereas parsing 100 POIs from JSON took less than three seconds.

A lesson learned from this is that web and especially Ajax development requires much more careful software design and performance considerations for mobile devices than for desktop computers. Our JSON-based version of the TELAR Mashup Platform is well-usable on the mobile device.

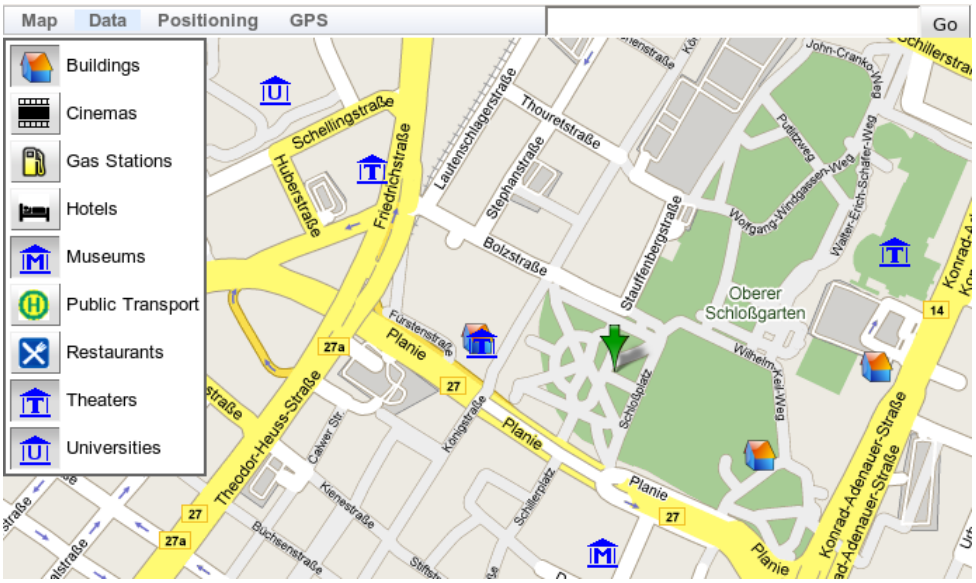


Figure 6.9: Screenshot of NexusWeb on a Nokia N810

6.3.2 NexusWeb

In [Brodt and Stach, 2009] we demonstrated a location-based mapping mashup system with a fundamentally different approach to data integration: NexusWeb. In contrast to the TELAR Mashup Platform, which is designed for integrating location-based data from arbitrary sources on the web, NexusWeb builds on top of the Nexus platform. The Nexus project aimed at a generic world model, the Augmented World Model (AWM) [Nicklas et al., 2001]. For this, Nexus developed the concept of a *World Wide Space*, an open, federated environment, to which arbitrary data providers may contribute local context models. The Nexus platform integrates these context models into a large-scale federated DMS using an object-oriented data model.

Applications built on top of the Nexus platform prior to NexusWeb were carried out as platform-dependent fat clients which had to be installed on the user's device. NexusWeb, by contrast, is a web application and requires only a web browser to work. If the browser is equipped with TELAR DCCI, NexusWeb is able to exploit local context information, but it is also usable without it.

Unlike the TELAR Mashup Platform, which lets the user to choose between different preconfigured data providers, NexusWeb offers control over the displayed POI data at a higher level. Building on the Nexus platform, NexusWeb exploits the AWM to abstract from single data providers. Instead, it offers the user a list of AWM classes to control the presented data objects, as shown in Figure 6.9 shows.

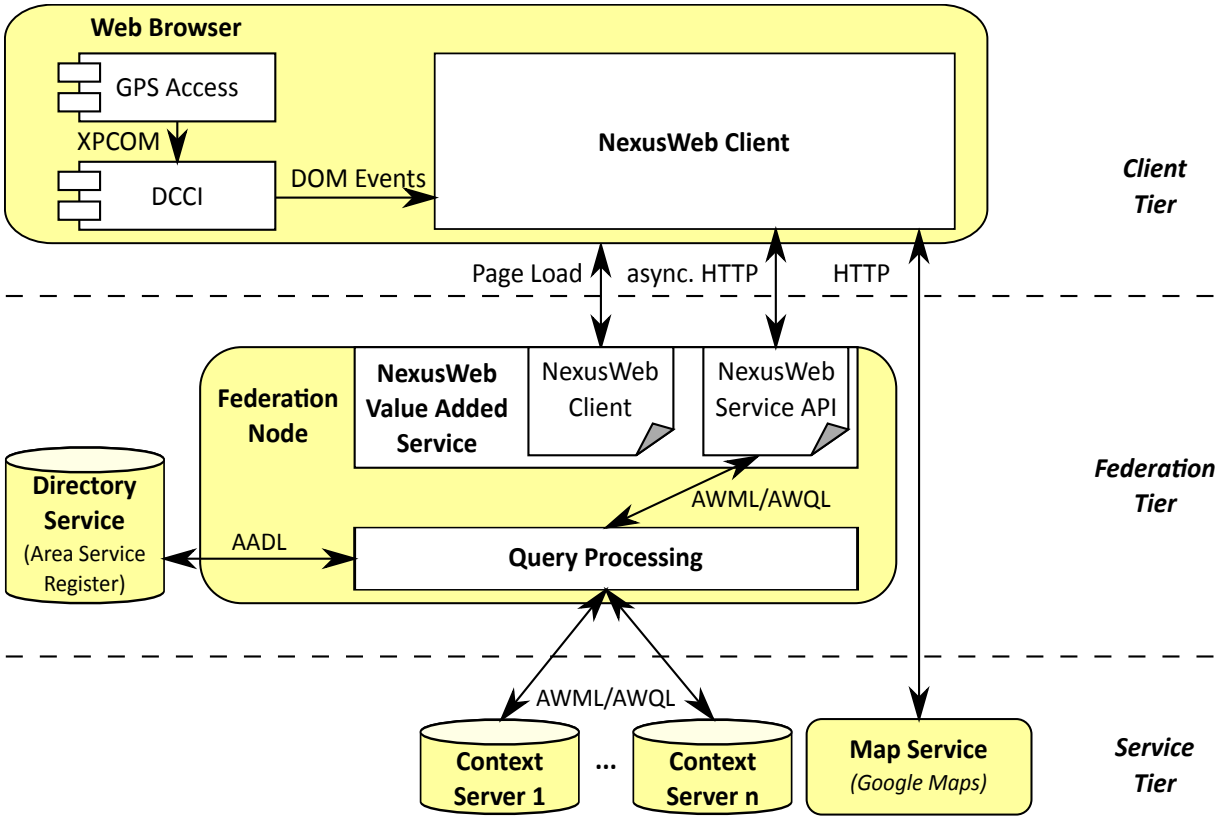


Figure 6.10: The system architecture of NexusWeb

Figure 6.10 illustrates the system architecture of NexusWeb. NexusWeb resides as a value-added service on a federation node of the Nexus platform, which is further described in [Nicklas et al., 2001]. NexusWeb consists of the NexusWeb client and the NexusWeb service API. The NexusWeb client contains the HTML, CSS and JavaScript files which make up the presentation and client-side logic of NexusWeb. The NexusWeb service API is an adaptor providing access to the Nexus federation.

As the TELAR Mashup Platform, also NexusWeb utilizes TELAR DCCI [Brodt, 2007b] to obtain the user's position. At startup the NexusWeb client registers an event listener at the DCCI module, to receive position updates from the GPS receiver of the mobile device. On position updates indicating a significant position change, the NexusWeb client loads additional objects from the Nexus platform. If DCCI or a GPS position is not available, the NexusWeb client still offers manual positioning.

To load additional objects from the World Wide Space, the NexusWeb client creates asynchronous HTTP requests to the NexusWeb service API. The NexusWeb service API translates them to the Augmented World Query Language (AWQL), the query language of Nexus, and forwards them to the Nexus query processor. The query

processor determines the context servers related to the query using the Area Service Register (ASR), a dedicated directory service. Them it forwards the request to the context servers in question. The query processor retrieves the results returned by the individual context servers by means of the Augmented World Modeling Language (AWML). It unifies the results, and returns them in a single AWML reply to the NexusWeb service API, which sends them to the NexusWeb client. The NexusWeb client visualizes the results by means of Google Maps.

6.4 Mobile Location-based Browser Games

Context-aware web applications are not only a beneficial technology for information systems. In [Brodt and Sathish, 2009], we applied this technology in the domain of mobile location-based browser games. Mobile location-based games are computer games in which players act by transmitting their movements in the real world to a virtual game world. Thus, mobile location-based games unite mobile games and computer games. Mobile location-based games integrate the real-world environment of players, including the people around them, into a virtual game experience and thus create a novel genre which augments reality.

BotFighters was one of the first commercial mobile location-based games [Sotamaa, 2002]. In BotFighters, the players, represented by robots, battle each other, but have to be physically within "shot distance". *Can you see me now?* [Flintham et al., 2003] and *Uncle Roy All Around You* [Benford et al., 2004] are games in which players at desktop computers have to interact and collaborate with mobile players moving in the real world. *REXplorer* is used for knowledge transfer; tourists equipped with smartphones and GPS receivers have to discover "strange phenomenons" in the historic city center of Regensburg, to learn about the town history [Ballagas et al., 2007]. In addition to their sophisticated game concept and content, all these games have in common that they require considerable technical effort, as programming of mobile devices is still strongly platform- and device-specific.

In the area of classic computer games, the recent years showed a trend towards browser games, i. e. computer games which use the web browser for the user interface. The first browser games used simple text-based user interfaces, which used the browser merely as a comfortable telnet client. In 1995 *SOL* [Spohr, 1995] was published as a browser game which supported multiple players and was capable of displaying a persistent game environment graphically. With an increase in professionalism of the browser games industry, *Fifth Season AS* published *Planetarion*, one of the first Massively Multiplayer Online Games (MMOGs) implemented as a browser game. As client-side scripting technologies and Flash emerged, many old computer games,

such as *Tetris*, *Pacman* or *Space Invaders*, were copied and could be played in the web browser spontaneously and without installation. Ajax also enabled sophisticated communication and visualization in the browser, which catered for large and complex browser games. Games including *Travianer* [Travian Games GmbH, 2008] are in many aspects comparable with classical installed computer games.

As browser games became more common in the web, they also caught the attention of existing websites. E. g. the social network *Facebook* opened an API to integrate arbitrary third-party web applications into Facebook. This is very commonly used for games. These games are thus instantly available to millions of potential players. Also, these games may incorporate personal data and social relations of players.

Browser games are easily and spontaneously accessible for players. Browser games may utilize and profit from web resources including images, maps, or social networks. Also, they may exploit the manifold communication methods of the web. Moreover, they are platform independent and thus accessible from any computer that is connected to the internet and features a tolerably state-of-the-art web browser. Thus, players may follow a MMOG from different places and computers. Being platform independent, browser games reduce development costs for game manufacturers. In addition to that, browser games run on the servers of the particular manufacturer which facilitates billing, and thus software piracy is not an issue, either: the players must register at the game servers.

The combination of browser games with context-aware web application technologies, as discussed in this chapter so far, supports implementing mobile location-based games in the web browser, which unites the game genres of mobile location-based games and browser games. *Mobile location-based browser games* bring the advantages of browser games in the area of mobile games and thus open new opportunities, which we examine in the following.

6.4.1 Examples for Mobile Location-based Browser Games

We implemented two mobile location-based browser games with which we exemplify this game genre: TREASURECACHE und *TicTacToe in Teams (T^4)*. TREASURECACHE is heavily inspired by geocaching-type games. It is integrated into the social network site Facebook and exploits its resources. T^4 is a location-based version of the pen-and-paper game TicTacToe that is played in two teams. We used a Nokia N810 Internet Tablet equipped with TELAR DCCI, as explained above, as our development platform for mobile context-aware web applications.

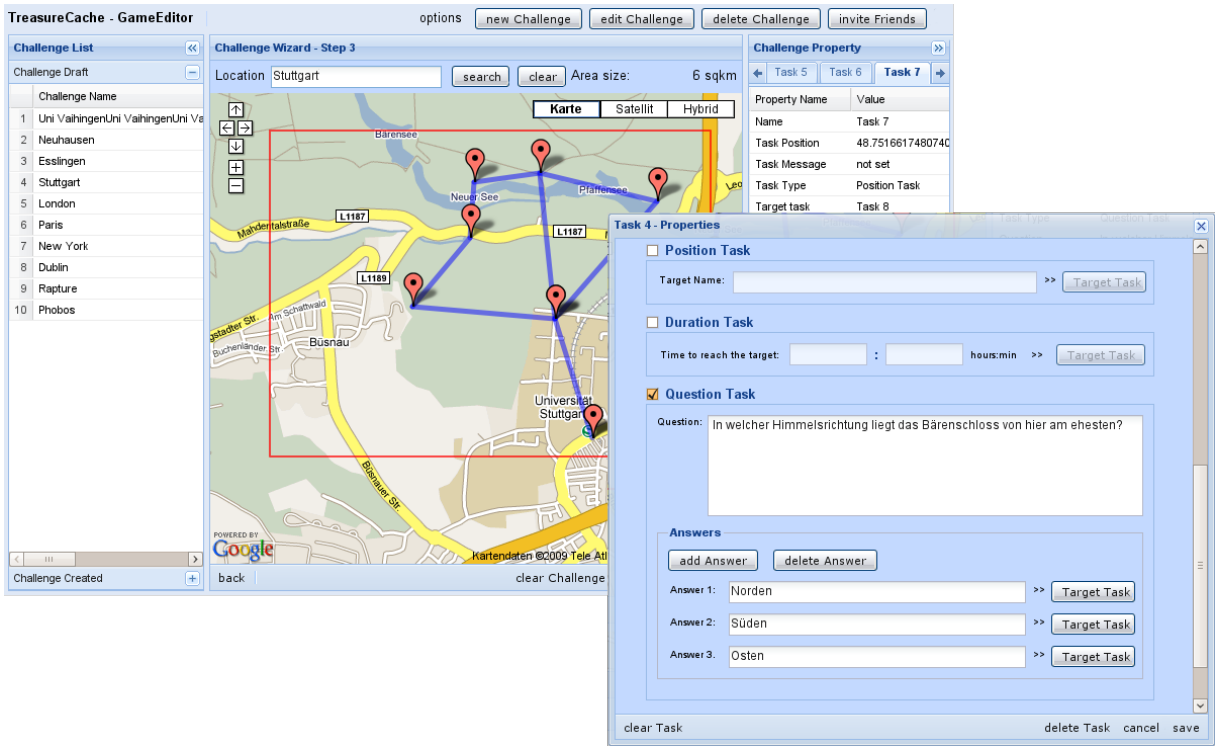


Figure 6.11: The TREASURECACHE Editor

TREASURECACHE

TREASURECACHE is a location-based browser game for Facebook. Its central concept are so-called *Challenges* which the player must accept and solve. A Facebook user creates such a Challenge using the web-based TREASURECACHE editor, which is depicted in Figure 6.11. Subsequently, the user may send the Challenge to her Facebook friends, who may decide to accept it. To do this, the player simply follows, on his mobile device, the web-link that is included in the invitation message of the Challenge. The link leads the player to the TREASURECACHE game page, which obtains the GPS position of the player and shows it on a map together with the Challenge itself. If the player is close to the area of the Challenge the game begins.

To solve a Challenge, several *Tasks* must be completed. The Tasks are located at a particular geographical location. There are different kinds of Tasks: *Position Tasks* simply need to be found. *Duration Tasks* add time pressure and must be reached within a given time frame. *Question Tasks* require the player to answer a multiple-choice question. Except for the final destination, a Task is followed by one or more successor Tasks. This creates a graph of Tasks along which the player has to navigate. The successor Tasks depend on how well the player completed the current task. By

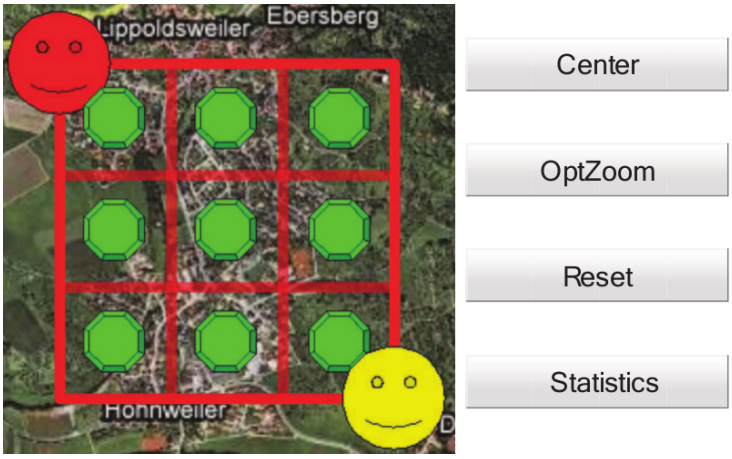


Figure 6.12: The user interface of Tic Tac Toe in Teams (T^4)

cleverly arranging the Tasks, the player can be directed to the final destination via a shorter or a longer path.

The player has mastered the Challenge when he has reached the final destination without any knock-out criteria, such as timeouts or wrongly answered questions. The TREASURECACHE game page measures the time which it took the player to complete the Challenge. In addition, the player may collect points when completing a Task well. The achieved result is published on the player's Facebook profile together with a ranking. Moreover, the created Challenges of a user are visible in the profile.

A basic concept of TREASURECACHE is that players must physically move to the geographic location of a Challenge, but not necessarily the creator of the Challenge. Challenges can be created using the TREASURECACHE editor and placed at arbitrary locations in the world. This allows challenging Facebook friends that live far away and fits the international character of social networks well. After all, one of the reasons that social networks are so popular is that they allow staying in contact with friends all over the world.

TicTacToe in Teams (T^4)

TicTacToe is one of the most well-known pen-and-paper games in the world. Two players occupy cells on a game board of 3 x 3 cells by turns, until one player occupied a full row, column, or diagonal line. *TicTacToe in Teams (T^4)* implements this game idea as a mobile location-based browser game. T^4 uses the geographic surroundings of the players as its game board and extends the number of players to eight players of two teams. Figure 6.12 shows the user interface of T^4.

As soon as eight players registered for a game on the T^4 website, they are divided into two teams and the game board is created. T^4 automatically determines the center point of the game board as the center of the initial player positions. The cells of the game board are arranged around the center point. This makes it possible to play T^4 anywhere without having to create the game board beforehand—of course the players should take care that the game board is accessible and does not contain dangerous areas. The starting positions of the teams are located in the opposite north west and south east corners. As soon as the teams have reached their starting positions the game begins.

The teams play by turns. A team first selects a free cell of the game board. This causes the cell to be *activated*, but not yet occupied. The team which first manages to send a player to the center point of the cell occupies the cell. I. e. if the opposite team is interested in the activated cell as well or if it wants to prevent the team from occupying the cell, the opposite team may send a player to the center point of the cell, too. If a player leaves the cell before the game is over, his team loses the cell. The cell may subsequently be activated and occupied again. A team may decide to give up a cell if it has no players left but not yet won the game. Also, the game state may have been changed, such that the cell is no longer tactically valuable for the team. Of course, a cell may be given up accidentally due to bad team coordination.

These tactical opportunities show that good coordination is essential for the teams; be it when activating a cell or when deciding which player to send to the cell. The teams must solve this on their own, e. g. via SMS, chat, or voice call.

6.4.2 Properties of Mobile Location-based Browser Games

Mobile location-based browser games, such as the examples presented in Section 6.4.1, bear a number of specific properties which affect their game concept as well as their technical implementation.

Influences on the Game Concept

Exploiting Web Resources A game that runs in the web can easily exploit the manifold resources that the web offers. For instance, TREASURECACHE utilizes Google Maps, in addition to Facebook users and their contacts. In addition, TREASURECACHE also utilizes the existing user authentication system of Facebook instead of implementing its own. Moreover, existing communication channels can be used. The players of TREASURECACHE may communicate with the creator of a Challenge via the Facebook chat or messages. As mobile location-based games take place in the real world, exploiting web resources about the surroundings of the player suggests itself.

Examples may include geographically annotated images or web articles. The (not browser-based) game *ECHOES* [Detken et al., 2008], for instance, utilizes the photo sharing web site *Flickr* to exchange images in the game environment.

Discontinuity As opposed to classical computer games, mobile location-based browser games cannot assume the player's uninterrupted attention. As it is generally the case with mobile games, environmental influences, such as street traffic, may force players to focus on other things. Also technical factors, e. g. an empty battery, bad GPS signal reception, or connectivity breakdowns may unintentionally interrupt or abort the game. Compared to a local application on a mobile device, it is more difficult for a web application to keep the game state. Local storage, e. g. as proposed in Section 6.2, may mitigate discontinuity from the technological side.

On the other hand, game developers must address discontinuity anyway, so that it is wise to design the game so that it supports interruptions in the first place. For example, a game could be split up into small episodes, such as the Tasks of TREASURECACHE. Also, a carefully chosen concept for saving the game state may be utilized in game design. Possibilities for this include, in addition to allowing to save the full game state any time, saving only at particular locations or points in time. Naturally, game design may deliberately chose not to allow saving at all.

Input Methods Mobile location-based games are played on mobile devices and input methods for mobile devices are simply not comparable to input methods of desktop computers. Also, game execution in the web browser further restricts input options. Thus, designers of mobile location-based games should ensure that the game can be played using simple user interaction. At the same time, mobile location-based games know the user's position and possibly further context information. In addition to that, they may observe the time. TREASURECACHE, for instance, solely requires the users to move to certain places and answer multiple-choice questions, which can be accomplished with few clicks. Similarly, the players of T^4 only need to activate cells and move to their center points.

Commnication Methods As the example of T^4 illustrates, team-based games require good coordination and thus suitable communication methods, which should be considered in the game concept. Real-time chats may be a suitable communication method, as well as the decision to omit communication methods deliberately, to make the game difficult and interesting. Mobile location-based games that are not browser-based bear the same requirement, e. g. as shown by *Can you see me now?* [Flintham et al., 2003].

A further aspect is that games are often successful because they are played by an active *community*. This should be supported through suitable communication channels, e. g. a forum. The web as a platform facilitates this considerably.

User-generated Content The *Web 2.0* trend [O'Reilly, 2005] strongly emphasizes user-generated content. Active computer players have always contributed to their favourite games, e. g. by adding further game characters, maps, or levels. A web-based game may make it particularly easy to add new content and share it with other players, e. g. by borrowing ideas from mashups, as discussed in Section 6.3.

Given a suitably active player community and easy ways of adding content, a mobile location-based game may easily reach geographical coverage that would be impossible to achieve by a team of game developers. However, the game concept must anticipate this [Wolff and Grüter, 2008]. The concept of the TREASURECACHE Challenges, for instance, is systematically targeted for user-generated content.

Technical Influences

Context Provisioning Mechanism Mobile location-based browser games are only made possible through a context provisioning mechanism for web applications. As stated in Section 6.4.1, we used TELAR DCCI to put our examples into practice. Until a widely supported context provisioning mechanism has been achieved, game developers must either choose a particular implementation that is sufficiently common in the targeted player community, or support multiple mechanisms.

User Interface As with input methods, means to create sophisticated graphical user interfaces are limited for browser games[2]. As shown in Section 6.3.1, the computing power of mobile devices is, despite all technological advances, still not up to desktop computers. Computationally intensive Ajax and Flash techniques may thus be problematic. Mobile location-based browser games should thus do without overly complex graphical effects. As the success of simple Flash games shows, games can be enormously fascinating even without sophisticated graphics. Mobile location-based games use the real world as their game board, so the graphical presentation is not a primary motivation to play the game.

[2]The game *Quake Live* (`www.quakelive.com`) achieves a sophisticated and complex 3D interface in the web browser through an application-specific browser extension. This, however, violates the concept of platform-independent web applications.

User Concept and Management Classical computer games and simple Flash games are typically simply started and played. A more complex browser game, by contrast, cannot do without a user concept that allows storing a user profile, game states, scores, and, not least, supports billing. In addition to typical user account management of web applications, browser games not infrequently need to manage teams. Thus, user management is required both at player level and at team level.

Depending on the game concept, game designers should consider adopting and, if required, extending existing user management systems. For instance, OpenID [OpenID Foundation, 2007] may be used for authentication and thus save development effort and increase security.

Use of Third-party Software As explained above, the web particularly facilitates integrating existing software components. Examples include communication methods, user management, or even multimedia components, such as mini-games or video clips. The architecture of the game should, however, anticipate this and provide sufficient flexibility. On the other hand, game developers must consider that third-party software may significantly influence the implementation. TREASURECACHE, for instance, utilizes user management and authentication as well as communication methods of Facebook. As a consequence, it cannot be played without a Facebook account. Also, changes in the Facebook APIs forced us to rewrite considerable parts of the implementation repeatedly.

Editor A graphical editor which allows creating game content in an easy way does not only facilitate game development. It can play an important role even after the game is published. If the game concept emphasizes user-generated content, as explained above, it is advisable to provide a graphical tool to enrich the game. A good example is the TREASURECACHE editor shown in Figure 6.11.

Persistence Without web browser extensions as discussed in Section 6.2, it is usually not possible for web applications to keep comprehensive data sets on the client side in a persistent manner. The game architecture must anticipate discontinuity, as stated above. Thus, it must ensure that all game data are retrieved from the web server and written back to there. At the same time it is advisable to keep large amount of game data on the client side, to make the game robust towards temporarily interrupted connectivity. TREASURECACHE, for instance, loads all Tasks of a Challenge immediately at startup. The discussed web browser extensions are capable of mitigating a fair amount of these issues by utilizing the Integrated Resource and Context Repository and the Repository Web-API.

6.5 Summary and Outlook

Interoperability with web applications is an important aspect in mobile data management scenarios, as the web is used from mobile devices to an increasing degree and large amounts of personal data are managed by web applications. We reviewed the foundations of web application technology from the beginning of the web to Ajax. Despite all progress of server-sided cloud storage, browser-local storage, domain-specific web-APIs, and context provisioning for web applications, a client-sided method to manage resource and context data in a way that makes the data accessible for other applications is missing. For this purpose, we proposed the Repository Web-API, which puts the web application interface of our architecture of Chapter 2 into practice. The Repository Web-API allows web applications to query and modify the Integrated Resource and Context Repository as well as to listen to events and to control selected resources. At the same time it features powerful access control based on access roles; the Repository Web-API executes all actions under a particular set of access roles. Furthermore, we addressed a particular subdomain of data interoperability with web applications: context-aware mashups. We presented the TELAR Mashup Platform, a web platform for context-aware mashups based on custom-written data wrappers. In addition, we presented NexusWeb, a location-based mapping mashup system which uses the Nexus federation for data integration. Finally, we applied our results in the domain of mobile location-based browser games, which offer a great show case for interoperability with web applications and greatly demonstrate the power of client-sided, context-aware, and data-interoperable web application technology.

Future Work

Interoperability of data that is locally available on the client side with web applications remains an interesting topic for future work. First and foremost, the Repository Web-API must be standardized as a general framework as well as the particular access roles, event types, controller interfaces and resource ontologies. This is a necessary, yet long and cumbersome work that involves many parties and goes beyond the capabilities of academic research. In the domain of web mashups, potential for future work lies in the sustainability of mashups, so that even in the case of data providers failing or changing their interfaces mashups can continue to provide user value. Finally, we expect a steep rise of mobile location-based browser games once the context and interporability interfaces are widely adopted.

7 Conclusions

We addressed interoperability at the data management level of mobile devices. As a large share of the data on mobile devices centers around the user's life, the data which mobile applications typically manage is highly interrelated and handled data domains overlap considerably between applications. However, this data is often kept separately and at best accessible via domain-specific APIs. Functionality to exchange data between co-located devices does not exist on the data management level, but must be implemented in application code. Finally, web applications, in which users keep large amounts of personal data, reside in a sandbox in the web browser, which excludes them from client-sided data management. This lack of interoperability creates data redundancy and ultimately impacts the user experience of mobile devices.

From this observation we identified four key requirements to achieve interoperability at the data management level of mobile devices:

1. A *data model* which is domain-independent, extensible, flexible, and capable of relecting well the interrelations between the data resources. It must support spatial data, as most of the data on mobile devices possesses spatial references directly or indirectly through the user's position. Finally, the data model must include an access control model.

2. An *integrated Data Management System* to keep the data of different applications in a single data model. It must support complex analytical queries as well as simple application-oriented queries. It must also support integrated spatial query processing to achieve spatial interoperability and it must perform access control.

3. *Ad-hoc inter-device connectivity* for interoperability between co-located devices. This includes a general concept, a suitable discovery mechanism that is robust towards the highly dynamic device neighorhood, and higher-level abstractions for applications to exploit resource and context data from co-located devices easily.

4. Browser-based data access for *web applications*, so that web applications may access resource and context data that is inherently client-sided. Also, given the respective access privileges, web applications should be able to contribute, such that other applications can again benefit web application data.

We presented a mobile data management architecture to enable interoperability between installed applications, co-located devices, as well as web applications at the data management level. Our approach is based on a central data repository on mobile devices which all applications use cooperatively. The data is stored and managed in an RDF-based data model and powerful access control that is tightly coupled with the data model regulates access to it. Subsequently, we made a number of essential contributions for central aspects of our architecture.

As a contribution for the integrated Data Management System, we presented an approach for *efficient record-oriented queries in RDF triple stores*. So far, the focus of most RDF query processors has been on finding complex graph patterns in RDF data. This typically involves a high number of joins. Obtaining a record-like view on the attributes of resources, by contrast, imposes unnecessary performance burdens in such query processing models. We proposed a novel processing model that exploits the existing RDF indexes of state-of-the-art triple stores in a different way. It splits queries to RDF data into *resource identification* and *attribute retrieval*. First, resource identification determines the resources of interest for the query. In a second step, attribute retrieval fetches the queried attributes in a single step for each identified resource. In addition, we proposed an index structure that is specifically designed for the access pattern of attribute retrieval. Our approach achieved clearly faster query times for typical application queries, which were further improved considerably by the dedicated index structure. This is important, as our architecture mandates all applications to manage their data in the same RDF database.

Furthermore, we addressed the *integration of spatial query processing into RDF triple stores*, which is a contribution for the integrated Data Management System as well. We presented a data and query model, an implementation approach, and a method for dedicated cardinality estimation of spatial RDF queries. We proposed to model spatial features as literal values of a complex data type. Likewise, we expressed spatial predicates as filter functions in SPARQL. Through this we managed to integrate

spatial query processing without extending the query language. To implement the query processing logic we added a spatial index as well as a spatial selection to the triple store architecture. To support the query optimizer create good query plans, we developed a cardinality estimation technique which divides the covered space into histogram buckets and bundles the most frequent paths adjacent to each bucket. This results in cardinality estimates that avoid assuming statistical independence to a large degree. In our evaluation we achieved fast spatial queries to large RDF data sets also on mobile devices.

To address ad-hoc inter-device connectivity we proposed the *concept of ad-hoc smart spaces* to enable spontaneous data exchange between mobile devices. As resource discovery in these highly dynamic device societies is a central challenge, we presented an evaluation of resource discovery protocols in Bluetooth-based ad-hoc smart spaces. Our simulations showed that for small network sizes Request Flooding performs best. Only for larger networks dedicated routing and replication structures pay off. We found the Random Replication protocol to cope with changes in the network best. In addition we presented two sample applications which are based on different interaction paradigms and well demonstrate the ad-hoc smart space concept also to non-technical users.

Finally, for interoperability with web applications, we specified a *web browser interface* via which web applications may access repository of our architecture: the Repository Web-API. In contrast to technologies such as server-sided "cloud storage", browser-local storage, domain-specific web-APIs, and context provisioning for web applications, the Repository Web-API provides a client-sided method to manage resource and context data in a way that makes the data accessible for other applications. To prevent misuse, it features powerful role-based access control and executes all actions under a particular set of roles. Furthermore, we addressed a particular subdomain of data interoperability with web applications: context-aware mashups. We presented the TELAR Mashup Platform, which integrates server-based data through custom-written data wrappers. In addition, we presented NexusWeb, which uses the Nexus federation for data integration. Finally, we applied our results on mobile location-based browser games, which greatly demonstrate the power of client-sided, context-aware, and data-interoperable web application technology.

Outlook

In this work we focused data management for mobile devices. We took a bottom-up approach exploiting the data in an integrated way to support a good user experience. Naturally, this must go with a corresponding user interface concept that builds on the integrated data model, as developed, e. g., by Lehikoinen et al. [2007]. This may require additional entry points to the data, e. g. a full text index, which can be added to our presented architecture in a similar fashion as the spatial index. In addition to that, we did, except for web mashups, not address synchronization with server-based "cloud" data and services. An actual consumer product offering should address this; however this strongly depends on the particular services that the offering provides, which is beyond the scope of this work.

Our contributions to RDF data management, our attribute retrieval approach as well as the deep integration of spatial query processing, are not restricted to mobile data management. They are, possibly with a different focus, applicable in general. As RDF triple stores appear to be an active and exciting research area for the near future, we expect new approaches to emerge in the areas of indexing, query processing, integration of complex data types, or cardinality estimation for RDF triple stores that may exploit our contributions.

Ad-hoc smart spaces are still a relatively new topic that is eagerly developed [Sathish, 2011]. With a broader uptake of the concept, new use cases and requirements will emerge. In the long term it may be worth developing new wireless communication technologies that are better suited for this kind of ad-hoc interaction than Bluetooth.

Finally, interoperability of web applications with data that is locally available on the client side remains an interesting topic for future work. Our contributions are capable of providing the baseline for this. With a wider adoption of these or similar approaches, we do expect a standard for this in the future.

List of Figures

List of Listings

BIBLIOGRAPHY

Daniel J. Abadi, Adam Marcus, Samuel R. Madden, and Kate Hollenbach. Scalable semantic web data management using vertical partitioning. In *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07, pages 411–422. VLDB Endowment, 2007.

Gregory D. Abowd, Anind K. Dey, Peter J. Brown, Nigel Davies, Mark Smith, and Pete Steggles. Towards a better understanding of context and context-awareness. In *Proceedings of the 1st international symposium on Handheld and Ubiquitous Computing*, HUC '99, pages 304–307. Springer-Verlag, 1999.

Swarup Acharya, Viswanath Poosala, and Sridhar Ramaswamy. Selectivity estimation in spatial databases. In *Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, SIGMOD '99, pages 13–24. ACM, 1999.

Apple Inc. Security overview. Mac OS X Reference Library, 2010. URL `http://developer.apple.com/library/mac/#documentation/Security/Conceptual/Security_Overview`.

Medha Atre, Vineet Chaoji, Mohammed J. Zaki, and James A. Hendler. Matrix "bit" loaded: a scalable lightweight join query processor for RDF data. In *WWW*, 2010.

Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. Dbpedia: a nucleus for a web of open data. In *ISWC/ASWC*, 2007.

Sören Auer, Jens Lehmann, and Sebastian Hellmann. LinkedGeoData: Adding a Spatial Dimension to the Web of Data. *The Semantic Web-ISWC 2009*, pages 731–746, 2009.

Simon Baatz, Matthias Frank, Carmen Kuhl, Peter Martini, and Christoph Scholz. Bluetooth scatternets: an enhanced adaptive scheduling scheme. In *INFOCOM 2002*.

Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE, volume 2, pages 782–790, 2002.

Rafael A. Ballagas, Sven G. Kratz, Jan Borchers, Eugen Yu, Steffen P. Walz, Claudia O. Fuhr, Ludger Hovestadt, and Martin Tann. Rexplorer: a mobile, pervasive spell-casting game for tourists. In *CHI '07: CHI '07 extended abstracts on Human factors in computing systems*, pages 1929–1934. ACM, 2007.

Martin Bauer, Christian Becker, and Kurt Rothermel. Location models from the perspective of context-aware applications and mobile ad hoc networks. *Personal and Ubiquitous Computing*, 6(5/6):322–328, 2002.

David Beckett and Tim Berners-Lee. Turtle – terse RDF triple language. Team submission, W3C, 2011. URL `http://www.w3.org/TeamSubmission/2011/SUBM-turtle-20110328/`.

Richard Beigel and Egemen Tanin. The geometry of browsing. In *LATIN'98: Theoretical Informatics*, volume 1380 of *Lecture Notes in Computer Science*, pages 331–340. Springer Berlin / Heidelberg, 1998.

Steve Benford, Martin Flintham, Adam Drozd, Rob Anastasi, Duncan Rowland, Nick Tandavanitj, Matt Adams, Ju Row-Farr, Amanda Oldroyd, and Jon Sutton. Uncle roy all around you: Implicating the city in a location-based performance. In *Proc Advanced Computer Entertainment at ACE 2004*. ACM Press, 2004.

Robin Berjon, Frederick Hirsch, Thomas Roessler, and Dominique Hazaël-Massieux. W3c device apis and policy working group, 2010. URL `http://www.w3.org/2009/dap/`.

Ansgar Bernardi. FP6027705 NEPOMUK Project Synopsis. Technical report, DFKI, 2008.

Tim Berners-Lee. Information management: A proposal. internal proposal, CERN, 1989. URL `http://www.w3.org/History/1989/proposal.html`.

Tim Berners-Lee. Worldwideweb: Summary. Newsgroup posting in alt.hypertext, 1991. URL `http://groups.google.com/group/alt.hypertext/msg/395f282a67a1916c`.

Tim Berners-Lee. World wide web, 1992.

Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific American*, 285(5):34–43, 2001.

Bluetooth Special Interest Group. Specification of the Bluetooth System, Core. Version 1.1, 2001.

Dan Brickley. Basic Geo (WGS84 lat/long) Vocabulary. W3C Semantic Web Interest Group, 2003. http://www.w3.org/2003/01/geo/.

Dan Brickley and Ramanathan V. Guha. RDF vocabulary description language 1.0: RDF schema. Recommendation, W3C, 2004. URL http://www.w3.org/TR/2007/CR-DPF-20071221/.

Andreas Brodt. Location-based mashups for nokia internet tablets. Diplomarbeit, Universität Stuttgart, 2007a.

Andreas Brodt. Telar DCCI website, 2007b. URL http://telardcci.garage.maemo.org.

Andreas Brodt and Nazario Cipriani. NexusWeb - eine kontextbasierte Webanwendung im World Wide Space. In GI, editor, *Datenbanksysteme in Business, Technologie und Web (BTW 2009), 13. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), Proceedings, 2.-6. März 2009, Münster, Germany*, volume 144 of *Lecture Notes in Informatics*, pages 588–591. GI, 2009.

Andreas Brodt and Daniela Nicklas. The telar mobile mashup platform for nokia internet tablets. In *EDBT '08: Proceedings of the 11th international conference on Extending database technology*, pages 700–704. ACM, 2008.

Andreas Brodt and Sailesh Sathish. Together we are strong— towards ad-hoc smart spaces. In *PERCOM '09: Proceedings of the 2009 IEEE International Conference on Pervasive Computing and Communications*, pages 1–4, 2009.

Andreas Brodt and Christoph Stach. Mobile ortsbasierte Browserspiele. In Gesellschaft für Informatik e.V., editor, *Tagungsband der 39. GI-Jahrestagung, 28.9. - 2.10.2009, Universität zu Lübeck*, Lecture Notes in Informatics. Gesellschaft für Informatik e.V. (GI), 2009.

Andreas Brodt, Daniela Nicklas, Sailesh Sathish, and Bernhard Mitschang. Context-aware mashups for mobile devices. In *International Conference on Web Information Systems Engineering (WISE)*, Lecture Notes in Computer Science. Springer-Verlag, 2008.

Andreas Brodt, Daniela Nicklas, and Bernhard Mitschang. Deep integration of spatial query processing into native RDF triple stores. In *Proceedings of the 18th SIGSPATIAL*

International Conference on Advances in Geographic Information Systems, GIS '10, pages 33–42. ACM, 2010a.

Andreas Brodt, Alexander Wobser, and Bernhard Mitschang. Resource discovery protocols for bluetooth-based ad-hoc smart spaces: Architectural considerations and protocol evaluation. In *Proceedings of the 2010 Eleventh International Conference on Mobile Data Management*, MDM '10, pages 145–150, 2010b.

Andreas Brodt, Oliver Schiller, and Bernhard Mitschang. Efficient resource attribute retrieval in RDF triple stores. In *Proceeding of the 20th ACM conference on Information and knowledge management*, CIKM '11. ACM, 2011a.

Andreas Brodt, Oliver Schiller, Sailesh Sathish, and Bernhard Mitschang. A mobile data management architecture for interoperability of resource and context data. In *Proceedings of the 2011 12th IEEE International Conference on Mobile Data Management*, MDM '11, pages 168–173, 2011b.

Barry Brumitt, Brian Meyers, John Krumm, Amanda Kern, and Steven A. Shafer. Easyliving: Technologies for intelligent environments. In *Proceedings of the 2nd international symposium on Handheld and Ubiquitous Computing*, pages 12–29, 2000.

Yatin Chawathe, Sylvia Ratnasamy, Lee Breslau, Nick Lanham, and Scott Shenker. Making gnutella-like p2p systems scalable. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '03, pages 407–418. ACM, 2003.

Luc Clement, Andrew Hately, Claus von Riegen, and Tony Rogers. Universal description, discovery, and integration (UDDI), 2008. URL `http://www.uddi.org/pubs/uddi_v3.htm`.

Douglas Crockford. The application/json media type for javascript object notation (json). Request for Comments 4627, The Internet Society, July 2006.

Conor Cunningham, César A. Galindo-Legaria, and Goetz Graefe. Pivot and unpivot: optimization and execution strategies in an rdbms. In *VLDB*, 2004.

Francesca Cuomo and Andrea Pugini. A linux based bluetooth scatternet formation kit: from design to performance results. In *Proc' REALMAN*, 2005.

Karen Detken, Carlos Martinez, Darren Carlson, Varvara Guljajeva, Mari-Klara Oja, and Andreas Schrader. Echoes - a crazy multiplayer pervasive game. In Heinz-Gerd

Hegering, Axel Lehmann, Hans Jï£irgen Ohlbach, and Christian Scheideler, editors, *GI Jahrestagung (1)*, volume 133 of *LNI*, pages 489–494. GI, 2008.

Anind K. Dey. Understanding and using context. *Personal and Ubiquitous Computing*, 5(1):4–7, 2001.

Björn Dick. Entwicklung eines Kostenmodells für den Optimierer einer nativen ortsbasierten RDF-Datenbank. Diplomarbeit, Universität Stuttgart, 2011.

Rob Ennals and David Gay. User-friendly functional programming for web mashups. In *ICFP '07: Proceedings of the 2007 ACM SIGPLAN international conference on Functional programming*, pages 223–234. ACM, 2007.

Orri Erling and Ivan Mikhailov. RDF support in the virtuoso DBMS. In Tassilo Pellegrini, Sören Auer, Klaus Tochtermann, and Sebastian Schaffert, editors, *Networked Knowledge - Networked Media*, volume 221 of *Studies in Computational Intelligence*, pages 7–24. Springer Berlin / Heidelberg, 2009.

Eva Fenrich, Andreas Brodt, and Daniela Nicklas. WODCA: A mobile, web-based field sampling support system. In *Proceedings of the 8th international conference on Hydroinformatics, HEIC 2009*, pages 1–10. International Water Association, 2009.

David F. Ferraiolo and D. Richard Kuhn. Role-based access control. In *Proceedings of the NIST-NSA National (USA) Computer Security Conference*, pages 554–563, 1992.

Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000. Chair-Richard N. Taylor.

Martin Flintham, Steve Benford, Rob Anastasi, Terry Hemmings, Andy Crabtree, Chris Greenhalgh, Nick Tandavanitj, Matt Adams, and Ju Row-Farr. Where on-line meets on the streets: experiences with mobile mixed reality games. In *CHI '03: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 569–576. ACM, 2003.

Ingbert R. Floyd, M. Cameron Jones, Dinesh Rathi, and Michael B. Twidale. Web mash-ups and patchwork prototyping: User-driven technological innovation with Web 2.0 and Open Source software. In *HICSS*, 2007.

Forum Nokia. Application signing, 2008. URL `http://wiki.forum.nokia.com/index.php/Application_Signing`.

Google Inc. Gears API, 2007. URL http://code.google.com/intl/de-DE/apis/gears/.

Google Inc. Android developer's guide, security and permissions, 2010. URL http://developer.android.com/guide/topics/security/security.html.

Götz Graefe. Volcano-an extensible and parallel query evaluation system. *Knowledge and Data Engineering, IEEE Transactions on*, 6(1):120 –135, 1994. ISSN 1041-4347.

Abdolbast Greede and Donal O'Mahony. A service driven routing protocol for bluetooth scatternets. In *Proc' Personal Mobile Communications Conference*, 2003.

Dimitrios Gunopulos, George Kollios, J. Tsotras, and Carlotta Domeniconi. Selectivity estimators for multidimensional range queries over real attributes. *The VLDB Journal*, 14:137–154, 2005. ISSN 1066-8888. doi: http://dx.doi.org/10.1007/s00778-003-0090-4. URL http://dx.doi.org/10.1007/s00778-003-0090-4.

Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57. ACM, 1984.

Eran Hammer-Lahav. The OAuth 1.0 protocol. Request for Comments 5849, Internet Engineering Task Force (IETF), 2010. URL http://www.ietf.org/rfc/rfc5849.txt.

Frank Harary. *Graph Theory*. Addison Wesley, 1969.

André. Herms and Michael Schulze. Publish/Subscribe Middleware für Selbstorganisierende Drahtlose Multi-Hop-Netzwerke. In *Selbstorganisierende, Adaptive, Kontextsens. vert. Sys.*, 2008.

John R. Herring. OpenGIS Implementation Specification for Geographic information - Simple feature access - Part 1: Common architecture. Candidate, Open Geospatial Consortium, Inc., 2006.

Ian Hickson. Web sql database. Editor's draft, W3C, 2010. URL http://dev.w3.org/html5/webdatabase/.

Ian Hickson. Web storage. Candidate recommendation, W3C, 2011. URL http://www.w3.org/TR/2011/CR-webstorage-20111208/.

Alex Hopmann. The story of XMLHTTP, 2007. URL http://www.alexhopmann.com/xmlhttp.htm.

Tim Howes and Mark Smith. *LDAP: programming directory-enabled applications with lightweight directory access protocol*. Macmillan Publishing Co., Inc., 1997.

IBM Inc. QEDWiki, 2007. URL `http://services.alphaworks.ibm.com/qedwiki`.

IBM Inc. IBM mashup center, 2009. URL `http://www-01.ibm.com/software/info/mashup-center/`.

Jena. Jena: a Semantic Web framework for Java. URL `http://jena.sourceforge.net/`.

Anant Jhingran. Enterprise information mashups: Integrating information, simply. In *VLDB*, 2006.

Ji Jin, Ning An, and Anand Sivasubramaniam. Analyzing range queries on spatial data. In *Proceedings of the 16th International Conference on Data Engineering*, pages 525–534. IEEE Computer Society, 2000.

Brad Johanson, Armando Fox, and Terry Winograd. The interactive workspaces project: experiences with ubiquitous computing rooms. *Pervasive Computing, IEEE*, 1(2):67 – 74, 2002. ISSN 1536-1268.

William Kammersell and Mike Dean. Conceptual search: Incorporating geospatial data into semantic queries. In *Terra Cognita – Directions to the Geospatial Semantic Web*, 2006.

Kapow Technologies Inc. Kapow software. URL `http://kapowsoftware.com/`.

Tim Kindberg, John Barton, Jeff Morgan, Gene Becker, Debbie Caswell, Philippe Debaty, Gita Gopal, Marcos Frid, Venky Krishnan, Howard Morris, John Schettino, Bill Serra, and Mirjana Spasojevic. People, places, things: web presence for the real world. *Mob. Netw. Appl.*, 7:365–376, 2002. ISSN 1383-469X.

Tor Klingberg and Raphael Manfredi. Gnutella 0.6, 2002. URL `http://rfc-gnutella.sf.net/src/rfc-0_6-draft.html`.

Graham Klyne and Jeremy J. Carroll. Resource Description Framework (RDF): Concepts and abstract syntax. Recommendation, W3C, 2004.

Dave Kolas. Supporting spatial semantics with sparql. *Transactions in GIS*, 12:5–18, 2008. ISSN 1467-9671.

Dave Kolas and Troy Self. Spatially augmented knowledge-base. In *ISWC+ASW*, 2007.

David M. Kristol and Lou Montulli. Http state management mechanism. Request for Comments 2109, IETF Network Working Group, 1997. URL `http://www.ietf.org/rfc/rfc2109.txt`.

Juha Lehikoinen, Antti Aaltonen, Pertti Huuskonen, and Ilkka Salminen. *Personal Content Experience: Managing Digital Life in the Mobile Age*. Wiley-Interscience, 2007.

Xuemin Lin, Qing Liu, Yidong Yuan, and Xiaofang Zhou. Multiscale histograms: summarizing topological relations in large spatial datasets. In *Proceedings of the 29th international conference on Very large data bases - Volume 29*, VLDB '2003, pages 814–825. VLDB Endowment, 2003.

Angela Maduko, Kemafor Anyanwu, Amit Sheth, and Paul Schliekelman. Graph summaries for subgraph frequency estimation. In *Proceedings of the 5th European semantic web conference on The semantic web: research and applications*, ESWC'08, pages 508–523. Springer-Verlag, 2008.

V. Markl, N. Megiddo, M. Kutsch, T. M. Tran, P. Haas, and U. Srivastava. Consistently estimating the selectivity of conjuncts of predicates. In *Proceedings of the 31st international conference on Very large data bases*, VLDB '05, pages 373–384. VLDB Endowment, 2005.

Christopher Martin. Making the internet of things happen. Personal discussion with Bosch Corporate Research, 2011.

Cameron McCormack. Web IDL. Working draft, W3C, 2011. URL `http://www.w3.org/TR/2011/WD-WebIDL-20110927/`.

Nikunj Mehta, Jonas Sicking, Eliot Graff, Andrei Popescu, and Jeremy Orlow. Indexed database api. Working draft, W3C, 2010. URL `http://www.w3.org/TR/IndexedDB/`.

Elena Meshkova, Janne Riihijärvi, Marina Petrova, and Petri Mähönen. A survey on resource discovery mechanisms, peer-to-peer and service discovery frameworks. *Comput. Netw.*, 52:2097–2128, 2008. ISSN 1389-1286.

Paul V. Mockapetris and Kevin J. Dunlap. Development of the domain name system. *SIGCOMM Comput. Commun. Rev.*, 18:123–133, 1988. ISSN 0146-4833.

Sudarshan Murthy, David Maier, and Lois Delcambre. Mash-o-matic. In *DocEng '06: Proceedings of the 2006 ACM symposium on Document engineering*, pages 205–214. ACM Press, 2006.

Antoni Mylka, Leo Sauermann, Michael Sintek, and Ludger van Elst. Nepomuk information element ontology. Technical report, OSCA Foundation, 2007. URL `http://www.semanticdesktop.org/ontologies/nie/`.

Lama Nachman, Ralph Kling, Robert Adler, Jonathan Huang, and Vincent Hummel. The Intel Mote® platform: a Bluetooth-based sensor network for industrial monitoring. In *Proceedings of the 4th international symposium on Information processing in sensor networks*, IPSN '05. IEEE Press, 2005.

Thomas Neumann and Guido Moerkotte. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In *ICDE*, 2011.

Thomas Neumann and Gerhard Weikum. RDF-3X: a RISC-style engine for RDF. *Proc. VLDB Endow.*, 1(1):647–659, 2008. ISSN 2150-8097.

Thomas Neumann and Gerhard Weikum. Scalable join processing on very large RDF graphs. In *SIGMOD*, 2009.

Thomas Neumann and Gerhard Weikum. The RDF-3X engine for scalable management of RDF data. *The VLDB Journal*, 19(1):91–113, 2010. ISSN 1066-8888.

Daniela Nicklas, Matthias Großmann, Thomas Schwarz, Steffen Volz, and Bernhard Mitschang. A model-based, open architecture for mobile, spatially aware applications. In *SSTD '01: Proceedings of the 7th International Symposium on Advances in Spatial and Temporal Databases*, pages 117–135. Springer-Verlag, 2001.

OpenGIS. OpenGIS Geography Markup Language (GML) Encoding Standard - Version 1.0.0, 2000.

OpenID Foundation. Openid, 2007. URL `http://openid.net/`.

Tim O'Reilly. What is web 2.0 - design patterns and business models for the next generation of software. web article, September 2005. URL `http://www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web-20.html`.

Matthew Perry, Farshad Hakimpour, and Amit Sheth. Analyzing theme, space, and time: an ontology-based approach. In *ACM GIS*, 2006.

Matthew S. Perry. *A framework to support spatial, temporal and thematic analytics over semantic web data*. PhD thesis, Wright State University, 2008.

Tom Pixley. Document object model (dom) level 2 events specification. Recommendation, W3C, 2000. URL http://www.w3.org/TR/DOM-Level-2-Events/.

Andrei Popescu. Geolocation API Specification. Editor's draft, W3C, 2009. URL http://dev.w3.org/geo/api/spec-source.html.

Eric Prud'hommeaux and Andy Seaborne. SPARQL query language for RDF. Recommendation, W3C, 2008. URL http://www.w3.org/TR/rdf-sparql-query/.

Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '01, pages 161–172. ACM, 2001.

Trygve Reenskaug. Models - views - controllers. technical note, Xerox PARC, 1979. URL http://heim.ifi.uio.no/~trygver/1979/mvc-2/1979-12-MVC.pdf.

Bastian Reitschuster. Entwurf und Implementierung einer mobilen Anwendung zur kollaborativen Terminplanung in Ad-hoc-Umgebung, 2010.

Jesse Ruderman. Same origin policy for javascript. Mozilla Developer Network, 2010. URL https://developer.mozilla.org/En/Same_origin_policy_for_JavaScript.

Sailesh Sathish. Motivation for ad-hoc smart spaces. Personal discussion with Nokia Research Center, 2011.

Leo Sauermann, Ansgar Bernardi, and Andreas Dengel. Overview and outlook on the semantic desktop. In *Proceedings of the 1st Workshop on The Semantic Desktop at the ISWC 2005 Conference*, 2005.

Leo Sauermann, Ludger van Elst, and Knud Möller. Personal information model (pimo). Recommendation, OSCA Foundation, 2009. URL http://www.semanticdesktop.org/ontologies/2007/11/01/pimo/.

Andy Seaborne, Geetha Manjunath, Chris Bizer, John Breslin, Souripriya Das, Ian Davis, Steve Harris, Kingsley Idehen, Olivier Corby, Kjetil Kjernsmo, and Benjamin Nowack. Sparql update. Member submission, W3C, 2008. URL http://www.w3.org/Submission/SPARQL-Update/.

Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond Lorie, and Thomas G. Price. Access path selection in a relational database management system. In *SIGMOD*, 1979.

SETI@home Project. Statistics and leaderboards, 2011. URL `http://setiathome.berkeley.edu/stats.php`.

Raj Singh, Andrew Turner, Mikel Maron, and Allan Doyle. GeoRSS: Geographically encoded objects for RSS feeds, 2009. `http://georss.org/gml`.

Olli Sotamaa. All the world's a botfighter stage: Notes on location-based multi-user gaming. In *Proceedings of the Computer Games and Digital Cultures Conference, June 6-8, 2002, Tampere, Finland*, pages 35–44, 2002. URL `http://www.digra.org/dl/db/05164.14477`.

Alexander Spohr. Sol, 1995. URL `http://www.freeport.de/Sol/`.

Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc' SIGCOMM*, 2001.

Fabian M. Suchanek, Gjergji Kasneci, and Gerhard Weikum. Yago: A large ontology from wikipedia and wordnet. *Web Semantics: Science, Services and Agents on the World Wide Web*, 6:203–217, 2008. ISSN 1570-8268.

Chengyu Sun, Nagender Bandi, Divyakant Agrawal, and Amr El Abbadi. Exploring spatial datasets with histograms. *Distrib. Parallel Databases*, 20:57–88, 2006. ISSN 0926-8782.

Richard Tibbett. Contacts api. Working draft, W3C, 2010. URL `http://www.w3.org/TR/2010/WD-contacts-api-20101209/`.

Richard Tibbett and Suresh Chitturi. The calendar api. Editor's draft, W3C, 2010. URL `http://dev.w3.org/2009/dap/calendar/`.

Dzung D. Tran, Ilkka Oksanen, and Ingmar Kliche. The media capture api. Working draft, W3C, 2010. URL `http://www.w3.org/TR/2010/WD-media-capture-api-20100928/`.

Travian Games GmbH. Travianer, 2008. URL `http://www.travianer.de/`.

Jilles van Gurp, Christian Prehofer, and Cristiano di Flora. Experiences with realizing smart space web service applications. In *Consumer Communications and Networking Conference, 2008. CCNC 2008. 5th IEEE*, pages 1171 –1175, 2008.

Keith Waters, Rafah A. Hosn, Dave Raggett, Sailesh Sathish, Matt Womer, Max Froumentin, and Rhys Lewis. Delivery Context: Client Interfaces (DCCI) 1.0. Candidate recommendation, W3C, 2007. URL `http://www.w3.org/TR/2007/CR-DPF-20071221/`.

Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. Hexastore: sextuple indexing for semantic web data management. In *VLDB*, 2008.

Alexander Wobser. bluescatnet project page, 2010. URL `http://code.google.com/p/bluescatnet/`.

Stephan Wolff and Barbara Grüter. Context, emergent game play and the mobile gamer as producer. In Heinz-Gerd Hegering, Axel Lehmann, Hans Jürgen Ohlbach, and Christian Scheideler, editors, *GI Jahrestagung (1)*, volume 133 of *LNI*, pages 495–500. GI, 2008.

Jeffrey Wong and Jason I. Hong. Making mashups with marmite: towards end-user programming for the web. In *CHI*, 2007.

Catharine M. Wyss and Edward L. Robertson. A formal characterization of pivot/unpivot. In *CIKM*, 2005.

Yahoo! Inc. Yahoo pipes, 2007. URL `http://pipes.yahoo.com`.

Arkady Zaslavsky. Incentives for ad-hoc smart spaces. Personal discussion at the Mobile Data Management Conference (MDM), 2010.

All web links were last followed on December 12, 2011.

Curriculum Vitae

Andreas Markus Brodt	
Date and place of birth:	February 14th, 1981; Gaildorf, Germany
Nationality:	German
11/2012	Final submission of this dissertation
01/2008 – 12/2011	Research staff member at the Institute of Parallel and Distributed Systems (IPVS), Universität Stuttgart, Germany
10/2001 – 10/2007	Studies in Software Engineering ("Softwaretechnik") at Universität Stuttgart, Germany Degree: Diplom-Informatiker (Dipl.–Inf.)
01/2007 – 08/2007	Diploma thesis at Nokia Multimedia in Oulu, Finland (continued until 12/2011)
03/2006 – 12/2006	Software Developer/Consultant at flexis AG in Stuttgart, Germany
06/2005 – 08/2005	Trainee at Nokia Technology Platforms in Oulu, Finland
08/2004 – 05/2005	ERASMUS exchange studies at University of Oulu, Finland
09/2000 – 08/2001	Alternative National Service at a school for mentally handicapped children in Schwäbisch Hall, Germany
09/1991 – 07/2000	Secondary School at Schenk-von-Limpurg Gymnasium in Gaildorf, Germany
09/1987 – 07/1991	Primary School at Stadtschule in Gaildorf, Germany