HOCHSCHULE KONSTANZ TECHNIK, WIRTSCHAFT UND GESTALTUNG
UNIVERSITY OF APPLIED SCIENCES

Migrating the XML Database BaseX to the Android Platform and Analyzing the Result

Stephan Hagios

Konstanz, 28.03.2014

MASTERARBEIT

MASTERARBEIT

zur Erlangung des akademischen Grades

Master of Science (M. Sc.)

an der

Hochschule Konstanz
Technik, Wirtschaft und Gestaltung

Fakultät Informatik
Studiengang Master of Science Informatik

Thema:	**Migrating the XML Database BaseX to the Android Platform and Analyzing the Result**
Masterkandidat:	Stephan Hagios, Zasiusstraße 17, 78462 Konstanz
1. Prüfer:	Prof. Dr. Oliver Eck
2. Prüfer:	Dr. Christian Grün
	Dr. Alexander Holupirek
Ausgabedatum:	28.09.2013
Abgabedatum:	28.03.2014

Zusammenfassung (Abstract)

Thema:	Migrating the XML Database BaseX to the Android Platform and Analyzing the Result
Masterkandidat:	Stephan Hagios
Firma:	BaseX GmbH
Betreuer:	Prof. Dr. Oliver Eck Dr. Christian Grün Dr. Alexander Holupirek
Abgabedatum:	28.03.2014
Schlagworte:	BaseX, XML, Database, Android, Performance, SQLite3

The present thesis outlines the migration of the XML database BaseX to the Android platform. BaseX is available for desktop platforms, therefore it is outlined how Android and those platforms are differ. All necessary steps to build BaseX on Android are described and the process of creating an Android library which offers all needed BaseX functionalities is shown. After having created the working Android library of BaseX, the performance of the database has been analyzed and improved. For this purpose, different benchmarks on various devices have been used to identify the bottlenecks of the mobile BaseX version. After identifying these problems, it is shown which of them are depending on a specific platform constraint and which can be improved by changing the software. The performance differences between BaseX and the default Android database SQLite3 are also illustrated and which of these should be prefered for a specific domain.

Ehrenwörtliche Erklärung

Hiermit erkläre ich *Stephan Hagios, geboren am 03.07.1985 in Freiburg, im Breisgau*, dass ich

(1) meine Masterarbeit mit dem Titel

Migrating the XML Database BaseX to the Android Platform and Analyzing the Result

bei der BaseX GmbH unter Anleitung von Prof. Dr. Oliver Eck selbständig und ohne fremde Hilfe angefertigt und keine anderen als die angeführten Hilfen benutzt habe;

(2) die Übernahme wörtlicher Zitate, von Tabellen, Zeichnungen, Bildern und Programmen aus der Literatur oder anderen Quellen (Internet) sowie die Verwendung der Gedanken anderer Autoren an den entsprechenden Stellen innerhalb der Arbeit gekennzeichnet habe.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Konstanz, 28.03.2014

(Unterschrift)

Contents

List of Figures

Listings

List of Abbreviations

ADT	Android Development Tools
API	Application Programming Interface
APK	Android Application Package
ART	Android Runtime
dex	Dalvik Executable
DOM	Document Object Model
DVM	Dalvik Virtual Machine
GUI	Graphical User Interface
HTML	Hypertext Markup Language
IPC	Inter-process Communication
JDK	Java Development Kit
JIT	Just In Time
JNI	Java Native Interface
JVM	Java Virtual Machine
SAX	Simple API for XML
SDK	Software Development Kit
VM	Virtual Machine
XML	Extensible Markup Language

Chapter 1

Introduction

In the last five years, since the release of the first Android phone in October 2008, a lot of progress in the field of mobile devices has been made. Looking at the market share in the beginning of 2009 shows an amount of 2.8% sold Android phones [17]. Compared to this, five years later, more than 80% of the sold smart phones are using Android as their operating system. This is only one aspect that illustrates the triumphal course the Android platform has made the last few years. In addition to this the number of available applications for Android phones has also increased and has reached more than a million in the third quarter of 2013. Google offers with its Android Play Store the possibility for every developer to easy distribute their applications and it provides the possibility to reach millions of customers world wide. This opportunity is responsible for the amount of new applications which are occurring every day and that they recently reached the index of more than one million. The chance to reach a wide audience with one application is not the only reason for this high number of available applications. The available developer tools and libraries for Android development are also responsible for its success, which are also being evolved through the last few years.

Compared to this the Extensible Markup Language (XML) is a much older and completely different technique, which has been also very successful since its development, which started in 1996, and its first release in 1998 [7]. XML can be described as a format to store data in a text file using a markup language. It is used for exchanging, storing and defining data, as well as it is extended to other document formats. Therefore, there are also a lot of developer tools or libraries that use or provide the possibility to work with XML. One of these is the XML database engine BaseX, which is also an XQuery processor and offers features like a full text search as well as a client/server architecture. The present thesis aims to receive a working Android version of the XML database BaseX and analyzing the received result by benchmarking and comparing it to an other database system.

1.1 Motivation

Smartphones and other mobile devices are gaining more popularity with every day and every sold device. There are various operating systems for such devices, but none of them is that successful as Android in the last few months. Besides the fact that Android is becoming more popular with every sold smartphone or tablet device, it is also getting more successful in other fields. For example there exists research about its abilities to be an operating system for embedded systems [25] [28]. Just as well as there exists research about Android as an operating system for board computers in newer cars, to provide navigation, entertainment and status information to the driver [27]. Looking at its triumphal procession in the last three years, it can be said that it will play an important role in the future as an operating system which can be used in many fields and not only in combination with smartphones.
The same applies for the storage format XML, which has been becoming more important in the last years, since its initial release in 1998. XML is used to define graphical interfaces in Android and its storage ability is being more distributed every day. One possible explanation for its success is the fact that it is possible to adjust XML to receive other dialects of it using an own syntax, how it has already been done for many other formats. As mentioned before XML is used for storing data and as a result XML databases have been developed, which are using XML as their format to store the data and representing it. Similar to the query language of relational database systems SQL there is also a language to query and process XML files, that is also been used in XML database and it is called XQuery.

Resulting of the mentioned facts, it is obvious that sooner or later an implementation of an XML database for Android and a corresponding XQuery/XPath processor is made, which is also not available for the Android operating system at the moment. Consequently the present thesis outlines a first approach to offer developers the ability to use an XML database and also an XQuery/XPath processor for their Android applications.

1.2 Overview

The present thesis describes the migration of the native XML database BaseX to the Android operating system. Therefore, first an outline is given, about the database, the Android operating system and if there exists relating work to this topic. The same section also emphasizes that there is no XML database available for Android, which underlines the purpose of the present thesis and its resulting implementation. In Chapter 3 the source and target platforms of BaseX have been analyzed as well as the internal dependencies which BaseX is using. After this the database has been migrated to the Android platform and a library has been created which provides the usual BaseX operations. In addition to the library the client/server architecture of BaseX has also been implemented for the Android platform. At the end of the chapter the problems and issues as well as their solutions, during the migration process, are illustrated. In the next chapter the ported version of BaseX has been analyzed with different techniques and potential bottlenecks have been identified. The Android library has been optimized by improving the found bottlenecks. It has been compared to the available SQLite3 database and a proposition has been made for what kind of use case which database system is the better choice. In the end of the this chapter the constraints of the BaseX Android version have been searched and explained. In the last chapter a summery is given as well as a conclusion and proposals for future work which could pick up where this thesis ends.

Chapter 2

Outline

In this chapter an overview of the used components is given. It also outlines the used technologies of the present thesis. First a short introduction into the XML database BaseX is given and then the mobile operating system Android is introduced. At the end of the chapter it is outlined if there is any related work available and how this is important or impacting the present thesis.

2.1 Introduction into BaseX

BaseX is a native XML database and an XQuery/XPath processor which has been originally developed at the University of Konstanz. An XML database is a database which stores or processes its data in the XML format, by using the functional programming and query language XQuery. It is developed as an open source project and it is currently available in version 7.8. BaseX offers many features, a graphical user interface and an XQuery editor, to just name a few. It is implemented using the programming language Java and is therefore platform independent, as well as there are many clients in different languages available. Since 2012 there is also the company BaseX GmbH[1] which is taking care of the further development and maintaining of it.
BaseX provides a full XQuery 3 processor and the first full-text implementation for XML documents [21]. XQuery is a query, as well as a functional programming, language for processing XML data [4]. To query data XQuery provides the so called FLOWR expressions, which are containing *FOR, LET, ORDER BY, WHERE* and *RETURN*. Those can be seen as complement to the SQL statements *SELECT, FROM and WHERE*.
The above mentioned XQuery Full-text extension, known as XQuery 1.0 and XPath 2.0 Full-Text (XQFT), is used to combine full-text queries with XQuery statements. It supports stemming, synonyms, case sensitivity and the consideration of stop words [1].
With its scalability and its rich feature set, BaseX is a state of the art XML database which is applied in many domains.

[1]`http://basex.org`

2.2 Overview of Android

Android is an operating system developed by the Open Handset Alliance[2] and it was first released in October 2008 [2]. It aims to be an operating system for mobile devices such as smartphones or tablet devices and it is designed to be used with a touchscreen. It is open source and currently available in its newest version 4.4.2, codename KitKat. Since its initial release in 2008, it is being continuously developed further and with every new release it solves more of the appearing and well known challenges of mobile computing [15]. Android is based on a Linux kernel and offers everything what is expected of a state of the art operating system, for example drivers for sound as well as high quality 2D or 3D graphics [8]. One of Androids key features is its big variety of supported processors and hardware, since the release of the Android version 3.0 also tablet devices are officially supported.
Additionally, it has been especially designed for mobile devices with a small amount of available hardware resources, to match the requirements of mobile computing [32]. To achieve this, an advanced process management system has been implemented which handles every active application and all processes that are running in the background. With this process management, which is autonomic and can not be affected by any application or a background service, Android tries to provide as much hardware resources as possible to an active application.
Android offers a lot of possibilities and available tools for application developers. Constitutive of the above mentioned Linux kernel for the Android platform there are libraries which are providing access to the drivers and support other crucial mechanisms. Those libraries, as well as a provided application framework, offer everything a developer needs to build applications for Android. This is one of the reasons why there are more than one million applications available in the Android store and why Android is also very attractive for developers.

2.3 Related Work

Android offers an SQLite3 database which can be accessed from inside an application by using Java libraries. SQLite3 is a relational database system especially designed for embedded devices with a low requirement of resources. It provides beneath all relevant SQL commands also other mechanisms like views or triggers [34]. With its low hardware requirement it is a perfect relational database system for the Android mobile operating system. But it also is the only database available for Android application developers and at the same time the one which is officially provided by the operating system. According to [24] there is an Android implementation of the BerkleyDB database, but it appears not very common, because this is the only reference found to this topic. Unlike SQLite3 the BerkleyDB is not a relational database, but it also aims to

[2]Corporation of 84 companies aiming to evolve open standards in the context of mobile computing.

be deployed on embedded or mobile devices. Another difference between both database systems is that SQLite3 is implemented in C and the Android version of BerkleyDB is developed by using the Java programming language. Lamb [24] claims that this is the reason for BerkleyDB being three times faster than the SQLite3 database, because there is no need to translate the Java commands to the native C library, using the Java Native Interface (JNI). Another approach, for providing an alternative to SQLite3, is the object-oriented database Perst [3], which also offers an Android port. Similar to the BerkleyDB Android version Perst is also implemented using the programming language Java and it targets the embedded devices sector, too. According to the benchmarks of [29] Perst is up to sixteen times faster than the SQLite3 database. For an application developer this could be a considerable alternative to the native SQLite3 database. Despite this statement those three databases are the only available ones for Android and with BerkleyDB and Perst not natively covered by Android, most applications are using the standard SQLite3 Android database for storing their data. Even if these three databases are all using different database models, an XML database is not available for Android. A port of BaseX to this platform would cover this gap and also provide another alternative to the already existing database systems.
BaseX is not only a database it also provides a XQuery processor to query XML documents. Android offers the possibility to work with XML documents for its applications developers. Therefore the Document Object Model (DOM) package, as a library inside the application framework, is available. DOM is an application programming interface (API) for XML documents which have to be well-formed, as well as for HTML documents [31]. As a result of this, it is possible to parse XML documents into documents represented as objects inside an Android application and access its nodes and attributes inside of this document. Android also provides an alternative to DOM XML parsing, the Simple API for XML (SAX) package which offers, like the name already suggests, classes to parse and process XML documents. Both XML processing mechanisms, provided by Android, have their advantages and disadvantages. It can be said that one advantage of DOM is that it is possible to access the documents after parsing them, because they are stored as objects inside the application heap. In contrast to this SAX creates events or callbacks while reading an XML document [37]. The price for storing the objects inside the memory is the decrease of execution time and the increase of resource consumption [9]. Since Android API version 8 the XML Path Language (XPath) is available to query and process the content of XML documents. Even if there are two possible ways to parse and one to query and process XML documents on the Android platform, there are not the FLOWR expressions, which are provided by XQuery, available.
Besides BaseX another XQuery processor implementation exits, MXquery[4] is one that also provides an Android library of its implementation. This implementation is presented on the Internet[5] and supports XQuery 3. No research

[3]`http://www.mcobject.com/android`

[4]`http://mxquery.org`

[5]`https://sites.google.com/site/mxqueryandroid/`

is currently available on this implementation of MXquery for Android and it seems that the development has stopped back in 2011.
This leads to the result that the field of native XML databases, supporting the newest XQuery 3 version, implemented and used on Android is not rudimentary explored right now. This also applies to the database systems currently available for use with Android. Considering the amount of alternatives, with only the BerklyDB Android implementation, to SQLite3 this is also a field that could be more opened up. Migrating BaseX to the Android platform hereby offers a new database system and an XML query processor supporting the newest version of XQuery.

Chapter 3

Porting BaseX to the Android Platform

In the present chapter it is shown how the BaseX database has been migrated to the Android platform. To achieve this goal the source and target platforms have been analyzed and all necessary requirements have been determined. The main focus lies on the different software platforms and not on the specific hardware aspects, due to the big variety of systems and devices able to execute both platforms. Therefore the two virtual machines and their execution byte formats have been compared and also the specific Android internal mechanisms have been explored. In Section 3.3 it is illustrated which parts of the BaseX version have been changed and in which way to receive a working BaseX Android version. The next section outlines the creation of a client/server architecture using BaseX on the Android operating system. The last section of the present chapter describes the problems which occurred during the migration process of the database to the Android platform.

3.1 Analyzing the Source and Target Platforms

This section outlines the two different platforms and their specific properties.The main part of it concentrates on the software part, because both platforms can be executed on a huge amount of hardware devices. The term platform is hereby defined as on one side the Java Virtual Machine (JVM) and on the other side the Dalvik Virtual Machine (Dalvik VM, or DVM) in the Android environment. For a better identification the JVM is marked as the source and the DVM as the target platform. In later sections the Android operating system is also termed as target platform.
The JVM and the DVM are both virtual machines (VM), but they vary in many different aspects. To clarify the term virtual machine it has to be said that a VM is a simulated computer, which can be a whole system with all parts a normal computer provides and also needs to execute a program. Or it is an abstraction layer which provides the functionality to execute a program on every system that runs the virtual machine. Unlike compiled machine code a program for a virtual machine is platform independent, because it does not matter on which

operating system it is executed, as long as the virtual machine is available for the specific operating system. A disadvantage of this is the loss of speed, which is a result of the execution of the virtual machine and not only the program in general [11].
For both virtual machines the programming language Java is used, which is being compiled into code that the VMs understand and are able to execute. Although both platforms are virtual machines they differ in some crucial ways. One of the main difference is that the JVM is a stack based and the DVM is a register based virtual machine. Another distinctness of both machines is that the DVM is optimized to be executed on mobile devices, meaning that it is designed to use less memory and having a low CPU usage than the JVM. Lesser hardware usage implies also lesser battery usage, which is also a factor which should be considered in the field of mobile development.
Another difference, which need to be considered, is the host system, which in case of the DVM only includes Android. And as opposed to this it could be Windows, Mac OS, or any Linux/Unix derivate as host system running a Java virtual machine. This circumstance has to be considered if external resources will be used, like writing or reading a file for example, which is different in every operating system.

3.1.1 Comparison of the two Virtual Machines

As mentioned in the section before both source and target platforms are using Java as its programming language. Compared to other general purpose programming languages, for example C++, Java is not being compiled into machine code. To execute Java code a virtual machine is required, that executes the compiled Java code. However, as in Section 3.1 explained both platforms have different virtual machines. They differ in many kinds which an application developer not sees but need to consider.
One of the most important differences is that the JVM is a stack and the DVM is a register based virtual machine. This difference helps the DVM to execute the same code in lesser operations than the JVM. This means that the DVM need lesser CPU cycles than the JVM, which is an improvement that is necessary due to the lack of CPU and memory resources in mobile devices and the aspect of battery usage. This can be demonstrated by comparing the instructions done by adding two integers in both virtual machines. The stack based virtual machine has to execute four machine instructions, while the register based VM is able to do the same operation in one single instruction. The reason for this is that the stack based VM needs to pop the two values first before it can add and store them back on the stack. The instructions included in the example operation for the stack based machine are *'pop 1', 'pop 2', 'add 1 2', 'push result'*.
Figure 3.1 illustrates the instruction steps which need to be performed to execute an addition with a stack based virtual machine.

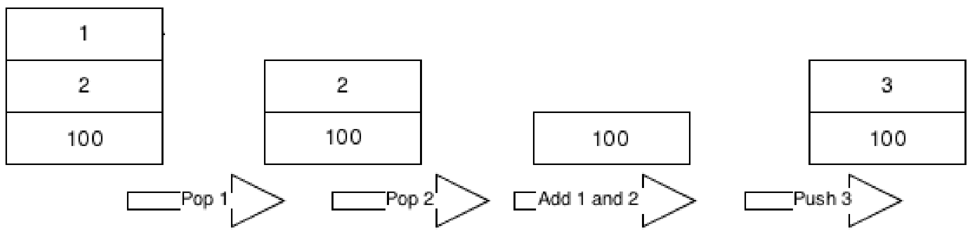


Figure 3.1: The addition of two integers on a stack based virtual machine.

Compared to this, the register based virtual machine just needs one single machine instruction to complete the same addition of two numbers. The required instruction for this task is *'ADD R1, R2, R3'*, which can be translated into: add the contents of register 1 and register 2 and store the result in register 3. Figure 3.2 illustrates this by showing the add operation of the numbers '1' and '2'.

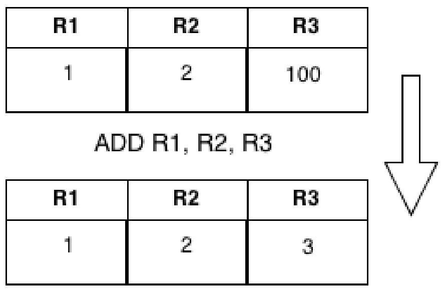


Figure 3.2: The addition of two integers on a register based virtual machine.

This minimal example shows how the register based virtual machine uses lesser machine instructions compared to the stack based. The disadvantage in this approach of a register based machine is that it has to store the addresses, meaning the corresponding registers, of the operands. Which is not necessary at a stack based machine because the stack pointer always directs to the actual operand using the Last In First Out (LIFO) principle. This leads to the result, that stack based code is smaller than the equivalent register based code. Although the Dalvik VM is a register based virtual machine, the executed byte code is not bigger than the equivalent JVM code [39].
This is the result of the different formats of the executable virtual machine code, which is optimized in the case of the Dalvik virtual machine and is illustrated later in Chapter 3.1.2.
The code that represents the instructions which are executed on Dalvik VM or the JVM is called bytecode. To execute bytecode instructions virtual machines having an interpreter which interprets all instructions and performs them. The difference to normal interpreted languages is, that the JVM or DVM interpreters do not need to check the syntax of a program. This has already been done by

the Java compiler which compiles the Java source code into so called class files. Even if it is faster than other interpreted languages, it is still slower than code that is compiled into a certain machine language and executed directly on the hardware. This so called machine code is faster because there is no layer between the executions and the hardware, which could slow down the execution [3].
There is a technique which can significantly improve a virtual machine in performance aspects by adding a Just In Time (JIT) compiler to it. In general it can be said, that a JIT compiler, used by a virtual machine, compiles heavy used code segments or very expensive calculations into a much faster machine code. There is a great number of different types of JIT compilers and how they work, the two used by the JVM or the DVM are:

- method-based
- trace-based

The trace-based method works by looking at the most executed code fragments, especially loops, and compiles them into native machine code.
The method-based JIT mechanism, in contrast to the trace-based, compiles whole methods, which are often used and expensive in execution time, into native machine code. Since the release of the Android version 2.2, release name Froyo, the Dalvik VM has received a JIT compiler additionally to the interpreter mechanism. According to Cheng [10] the implemented Dalvik JIT compiler can speed up the execution of intensive operations up to five times. The Dalvik VM uses the above mentioned mechanism of a trace-based JIT compiler mixed with its usual interpreter. The given advantage of the trace-based method is that not whole methods are being compiled into machine code. Instead only the parts which are often executed are being compiled. This reduces the size of the code which needs to be compiled by the JIT and also omits the not so often executed methods parts like exception handling. It is needles to compile such parts in machine code to speed them up, because most of the execution time of the program they are not being called. This should also reduce the compile time of the JIT, because of the selective compilation into machine code.
To realize this the Dalvik virtual machine has received an additional thread, which is responsible for the JIT compilation. Beneath this new thread there is the main thread, that includes the interpreter. This interpreter interprets the bytecode and records the traces and their occurrences. If the amount of the occurrence is higher than a predefined number, the trace is being stored into the trace queue. The new added thread, the JIT thread, compiles the traces from the queue and writes the machine code into the code cache. The main thread now is doing a lookup if the bytecode that needs to be executed is available in the code cache, at the moment. If this condition is met, it uses the machine code from the code cache instead of interpreting the actual bytecode [33]. Picture 3.3 illustrates the principle of the two threads inside the DVM.

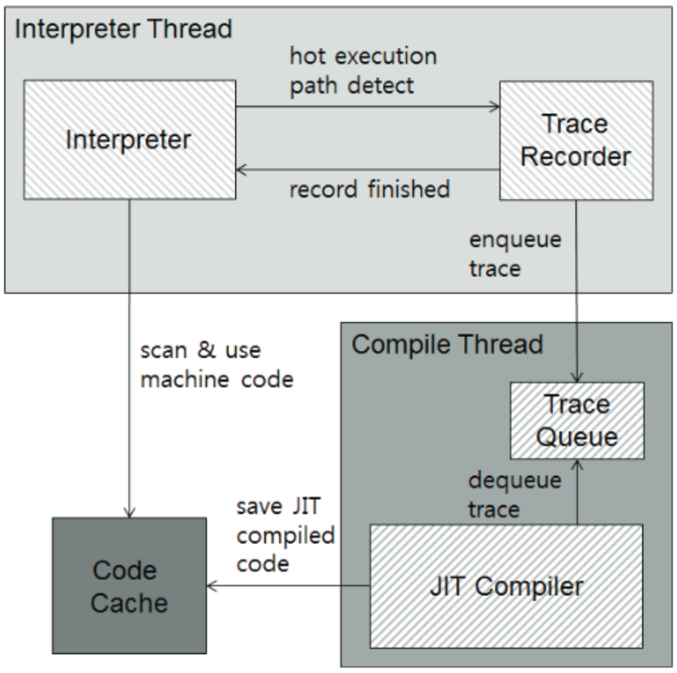


Figure 3.3: The principle of the two DVM threads. Source:[33]

Depending on the different implementation of a JVM, it differs in the way which type of JIT compiler mechanism is used. For this thesis the Oracle JVM *HotSpot* is used, which has a method-based JIT compiler as default. But it is also possible to use another JIT mechanism in the *HotSpot* JVM triggered by parameters during the VM start. The method-based JIT works as mentioned above, it compiles the most called methods into machine code [23]. Those methods are called hot spots, which are also responsible for the name of the virtual machine [35].
Both virtual machines are providing a garbage collector, which is responsible for the memory management of the executed bytecode. Same as the principle of an own instance of the Dalvik virtual machine per process, which will be explained in Section 3.1.4, every process also has its on instance of a garbage collector on Android. It is a mark-and-sweep garbage collector that only collects the objects that are not referenced any more inside the application heap. The shared heap, which is part of the Zygote process, and which is also described in Section 3.1.4, has its own garbage collector [28].
Although both virtual machines are using Java as programming language, Dalvik lacks of some libraries that are available for the JVM and vice versa. In Section 3.2 the libraries used by the BaseX database are analyzed and investigated which of them are supported by the DVM.
An overall statement about both virtual machines could not be made, because

the fact that they differ in many aspects. Especially considering that the DVM has been created for the mobile context and can only be ran on Android devices[1].

3.1.2 Comparison of the two Bytecode Formats

Both virtual machines also differ in their usage of formats for the executable files. On one side there is the Java Archive File (JAR) for the JVM and on the other side the Dalvik Executable (*dex*). The *dex* is being packed into an Android Application package file (*apk*) which can be executed by the Android operating system. A JAR file is an archive which includes the compressed class files. These class files are being build out of the Java source code by using the *javac* compiler [36]. An *apk* file is also an archive file, but it does not only include the executable program, it also includes the meta information and resource files. Although an Android application is an *apk* file only the format of the *dex* file is outlined here, due to the fact that the DVM executes those files. A *dex* file is created by using a tool which compiles Java class files into *dex* files. This tool is called *dx* and it is part of the Android software development kit. Picture 3.4 illustrates the flow of creating an *apk* file, which differs from the creation of a JAR file by creating the *dex* files out of the class files before packing them into an archive.

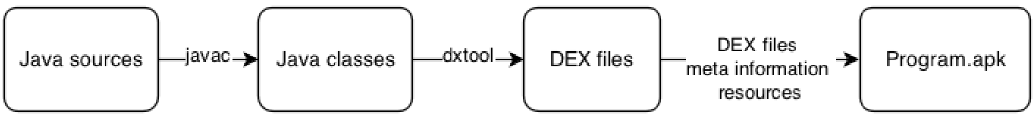


Figure 3.4: The creation flow of an *apk* file.

Having a closer look at the *dex* files and comparing them to the JAR files shows that they differ in various ways. The first thing that comes in mind is that a JAR file contains a class file for every Java class. Compared to this a *dex* file combines all specific information into one field. This is realized by just having one constant pool, in which all constant values of all classes are stored. These constants consist of:

string_ids Sorted list of all string identifiers

type_ids Sorted list of all class identifiers, arrays or primitive data types

proto_ids Sorted list of all prototypes

field_ids Sorted list of all identifiers for the used fields

methods_ids Sorted list of all methods used by the *dex* file

[1]Some community projects exist which are aiming to port the Dalvik VM to other platforms like Linux x86. For example the Android x86 project: `http://www.android-x86.org/`

Considering an interface which is used very often in a project can outline what advantage is given with the use of one shared constant pool. Looking at Image 3.5 demonstrates that every class which uses this interface has to have a reference within its own constant pool only and not the whole constant pool as itself.

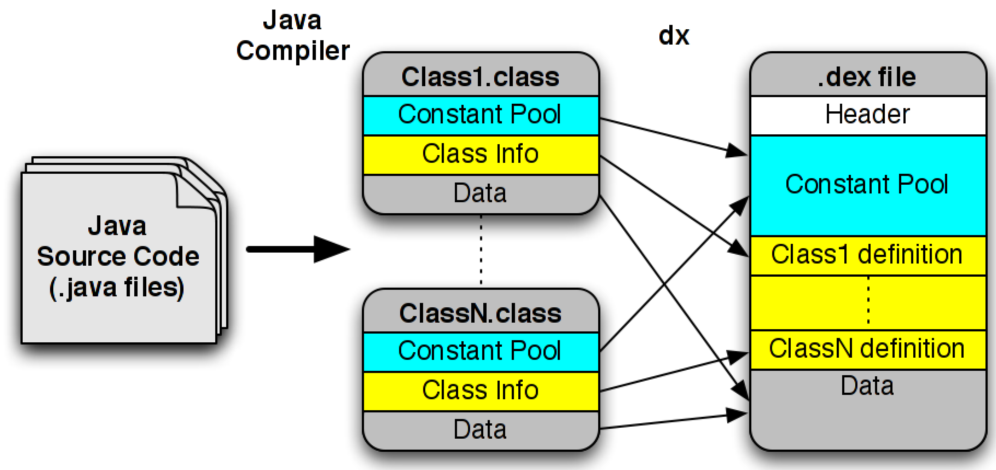


Figure 3.5: The difference between a jar and a *dex* file. Source:[14]

In general it can be said that *dex* files are smaller than their equivalent JAR files, because of using this shared constant pool [5]. An advantage of the creation chain of a *dex* file is that it is theoretically possible to create a *dex* file out of every jar archive. The problem hereby is that Android does not support all Java libraries that the JVM supports, but this can be passed over by using the above mentioned dx-tool. As already said before, a *dex* file is executed by the Dalvik virtual machine, but it is not the Android application. An Android application is the *apk* file that also includes user interface related information and other resource files, like images or sound files.

3.1.3 BaseX Internals

Although BaseX can be executed on every system that offers an implementation of the Java virtual machine, it has its specialties depending on the operating system that is used. This is the case if there is something about storing files on the file system. BaseX stores its databases and configuration files in a folder in the home directory of the user. This folder is created by the database if it does not exist yet.
Hence, BaseX has to figure out on which operating system it is running. After BaseX knows of the underlying operating system it stores several operating system specific information inside a static class, which is called *Prop*. This class needs to be adjusted during the migration process. One value inside this class is used to indicate if the path name is case sensitive, which is only relevant

for a Linux or Unix operating system and not for Windows or Mac. To store its databases inside the folder in the directory of the home directory of the current user BaseX needs the absolute path to this specific folder. Therefore the static Java system method *System.getProperty("user.home")*, which returns the absolute path to the home directory of the user on a normal JVM, is used. Another difference between Windows and other Unix based operating systems is that Windows uses a backslash "\" as path separator instead of a usual slash. This information is also stored inside the above mentioned *Prop* class by using the static Java system method *System.getProperty("line.separator").*

3.1.4 Android Internals

As shown in the Sections 3.1.1 and 3.1.2 the two virtual machines and their corresponding bytecode formats differ. However, Android has some other specialties about its internal processing procedure. The Android operating system is a Linux based operating system aimed to run on mobile devices. Therefore it has been created as a Linux fork of the 2.6.*[2] kernel and it has been especially designed for the mobile context. This includes a special focus on the resource consumption, because most mobile devices do not provide the same resources as a normal desktop PC or a notebook does. As shown in Section 3.1.1 an Android application is executed in the Dalvik virtual machine. At this point it has to be said that every application runs its own instance of the Dalvik VM, by forking the main Dalvik instance into an own Linux process. Here the Android policy sets in, where every application is independent from any other application and is executed in its own process, which is called sandbox. This principle also applies for all locations of files the application needs and uses. An application is similar in its rights and processing to a user in the Linux operating system. Every application is its own user and uses its own home directory. This directory can only be accessed by the owning application and is located in the /data/data system directory. This system directory can include its own SQLite3 database, cache directory, shared preferences[3] and every type of private data. The aspect of security is the reason for this type of solution as well as the stability of the application, considering the own Dalvik instance. This is called the *principle of least privilege* which specifies that privileges as less as possible are given to the application. Therefore application developers have to specify which privileges their application need, for example Internet access or the possibility to write on the external SD-card. It is possible that applications can communicate with predefined Inter-process Communications (IPC) but it is not possible to manipulate other applications. This can be omitted by granting root privileges to an application, but usually this is not possible on normal distributed devices. Hence, confidential data should also be encrypted, because theoretically it is possible to access the data inside the private directory. It also need to be said that every operation which reads or writes to one of the above mentioned directories or files is an input output (I/O) operation. This is always expensive in time consumption and should be done as little as possible. Image 3.6 illustrates

[2]Since Android version 4.* a Linux kernel 3.* is used

[3]Android framework that provides a mechanism to store primitive data types.

the principle that every application works in its own sandbox.

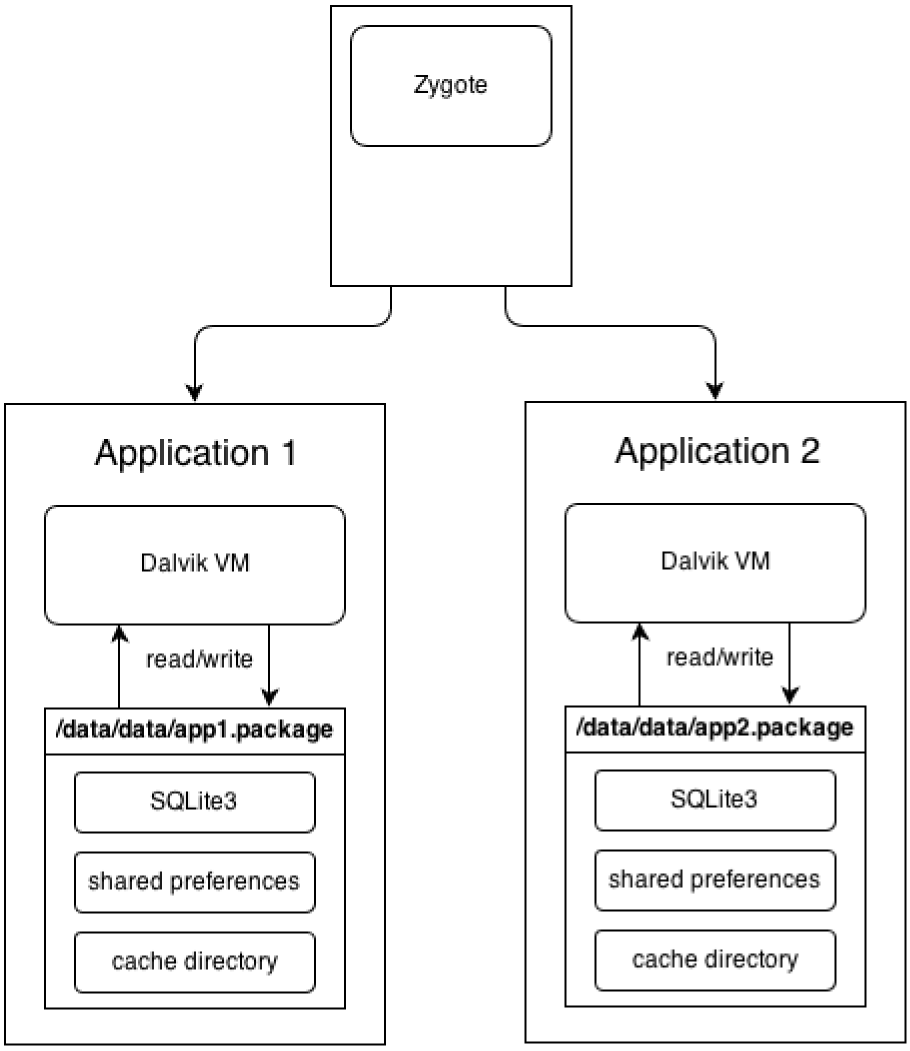


Figure 3.6: Illustration of the Android application architecture

The main Dalvik instance, which is the root parent for all forked virtual machines, is called Zygote and it is started by the boot up process of Android. Zygote is designed to load and initialize the core library classes so that every forked instance does not have to do this on its own. This is compared to starting a new virtual machine a speed improvement and saves memory, because not every VM instance has to have its own core libraries loaded. This can be done because the core libraries are all read only and are shared by all Dalvik instances [13]. Another difference to the usual Linux operating system is that Android uses the *Bionic libc* which is a special from Google developed library for Android, which has its own *pthread* implementation. This library is especially designed for the mobile context and the Android environment which improves the speed of forking the Dalvik virtual machine too. [6]

3.2 Migration of BaseX to Android

After analyzing both platforms the BaseX database code also needs to be examined. Like it has been mentioned in Section 3.1.1 the Dalvik virtual machine does not support all class libraries that are supported by the JVM. BaseX is written in the Java programming language and aims to be executed on virtual machines that are implemented by using the JVM specifications. Therefore it is not possible to transform the BaseX jar file into a *dex* file, using the dx-tool, and execute it on an Android device. On the other hand the DVM offers class libraries that the JVM does not support or is really aware of. These libraries do not intent to replace not supported Java class libraries. Their solely purpose is to provide special Android related methods, like logging or tracing of method calls as well as accessing hardware components of a mobile device, sensors or the mobile network for example.

3.2.1 Analyzing the Libraries used by BaseX

BaseX is a big project which consists of up to 60 packages and more than 1300 classes, interfaces and abstract classes. For this reason the examining of the used libraries and the dependencies a special tool was used: the so called Class Dependency Analyzer (CDA)[4]. It offers all needed possibilities for the mentioned task.
Parsing the whole BaseX project lists the used Java class libraries and external libraries. This list consists of 54 packages that are part of the Java class library or an external library used by BaseX. It has to be supposed that none of them are known to the Dalvik virtual machine.
Most of them represent subpackages of given packages and it can be expected that if a main package is not supported by Android its subpackages are neither. This assumption reduces the external packages to a total number of 16, which is compared to the before mentioned 54 packages an easier to analyze amount. Looking at the Android documentation the main packages that are not supported are just the two graphical user interface (GUI) related packages:

- java.awt[5] and
- javax.swing

All other packages are fully or at least partially supported by the Java language of the Dalvik VM. The reason why the packages awt and swing are not supported is that Android uses its own GUI libraries and framework. The Android provided GUI elements can either be written as Java code or can be defined in specific XML layout files, which get used to created the corresponding Java files during

[4]http://www.dependency-analyzer.org/

[5]Except the subpackage java.awt.font that provides two classes: NumericShaper and TextAttribute.

the compile process. The framework offers everything that is needed to build an application with a graphical user interface.
Supporting the other main packages does not mean that all subpackages, classes or even included methods that are used from other libraries are also known by the DVM. Therefore more investigation needs to be done in order to get a better overview of the supported packages. To get a clearer statement about the given situation, the subpackages are also being checked if they are supported by the DVM or not. The result of this investigation shows that javax.xml is supported, but not the following subpackages:

- javax.xml.crypto.dom
- javax.xml.crypto.dsig.dom
- javax.xml.crypto.dsig.keyinfo
- javax.xml.crypto.dsig.spec

Even if all packages and classes of javax.xml.crypto are part of the Java standard edition they are not available for Android development. All classes inside this packages are only used by one single BaseX package, which is called org.basex.query.util.crypto. This package is used to implement the XQuery cryptographic module that provides functionalities to en- and decrypt, sign and validate signed XML data. The only class that makes use of this package is the FNCrypto class. This class is used to provide the cryptographic module functionalities in XQuery. These are XQuery functions which consist of:

- hmac(string,string,string[,string])
- encrypt(string,string,string,string)
- decrypt(string,string,string,string)
- generate-signature(node,string,string,string,string,string[,item][,item])
- validate-signature(node)

To provide these five XQuery functions for Android, the javax.xml.crypto package needs to be migrated to Android, too. Another possibility to support these XQuery functions would be the use of other cryptographic classes which are provided by Android.
All other subpackages that are not directly mentioned are supported by the Android platform. For clarification not all methods that are provided by every used library are checked whether they are available for Android. This process would be to much effort, considering the high amount of available classes. Nevertheless this will be figured out in Section 3.2.3 and Section 3.3 outlines the methods which are not supported by different versions of Android.

3.2.2 The Android Project Structure

For developing Android applications an Android project needs to be set up initially. An Android project is a structure of special folders and predefined files, like code, resource and build files. There are three types of Android projects, which are differing in their functionalities and intentions.
The first is the type of project that needs to be created if an executable application is the goal of the development process. As a result of this an installable *apk* file will be created during the build process.
The second type of Android project specifies the so called Test project. This type of project aims to test executable Android projects by providing an Android testing framework including unit test, using the Java unit test framework *JUnit*. Both projects have the identical structure of folders and files.
The third project type represents the library project which goal it is to be a library that can be used by every other Android project. The difference to a normal Android project lies in the fact that this project is not being build into an *apk* file. Instead it is being used by other Android projects which pull it in their respective *apk* file and build it with their code. Depending on the specific Android API level an Android library differs from a usual Java library, by the fact that the Android library is not compressed into a jar archive. This feature has been added in Android versions higher than API level 14(Android 4.0 codename Icecream Sandwich). For older versions, before this API level, the library files were pulled into the project and compiled into the corresponding *dex* file.
If a new Android application is developed it is not necessary to set up the right project structure by hand. Therefore Android provides a Software Development Kit (SDK). This SDK provides tools and utilities to create each of the three types of a Android project. It also includes build, debug, trace and test tools. With this SDK, which is also called Android Developer Tools (ADT), it is possible to create Android applications written in Java. As it was mentioned in Section 3.1.2, the code that is translated into bytecode for the Dalvik virtual machine is written in the Java programming language. Though, it is also possible to write code in C++ and compile it into native code, like it is done by the, referred in Section 3.1.1, JIT mechanism. Therefore Google provides an Android Native Development Kit (NDK), that offers tools and a specific compiler to translate such native C++ code. If a project has been successfully created there are seven given folders, shown in Table 3.1.

Folder	Content
src/	The Java source code files
bin/	The compiled output file
jni/	C++ native code created by using the NDK
gen/	Automatically generated Java files
assets/	Every kind of file, which needs to be packed into the *apk* file as it is
res/	All XML resource files
libs/	The Private libraries

Table 3.1: Android project folders generated by the SDK.

The files that will be generated while creating an Android project consists of four build files that each holds information used by the build system to create the *dex* files and the *apk* archive. Another file that is being created is the so called *AndroidManifest.xml.* This XML file is important for the application and the Android operating system. It is used to specify the API level, the application name and other specific information related to the application. In this file it is also possible to grant privileges to the application that are needed for specific functionalities. For example accessing the Internet or writing on an external storage card. Libraries used within the project are also defined in this file. But if a referenced library is in need of specific privileges and defines them inside its own mainfest file, they are not automatically granted by Android. Therefore it is necessary to manually grant these privileges in the Android project and not in the library project, because declarations made inside a library manifest are being ignored.

3.2.3 Creating a BaseX Android Library

For the migration of BaseX to the Android operating system a library project has been created. The reason to make BaseX as Android library, as explained in Section 3.2.2, is that Android libraries can be used by many other projects. After creating the library project all BaseX files have been copied to this project. After this operation the GUI packages have been removed, those packages are basex.gui and all subpackages of it. As a result the depending classes of the GUI have also been removed. Those are also not necessary because the BaseX Android library does not need to provide a graphical user interface. Afterwards the not supported javax.xml.crypto packages and the dependent class FNCrypto have been removed. Therefore it not possible to use the cryptological XQuery function in the Android version of BaseX.
The result of removing all not supported packages, classes and libraries leads to the fact that it is possible to compile BaseX to an Android *dex* file, at this point. Although it is not possible at this moment to execute BaseX, because of some constraints, mentioned in Section 3.1.4, and not facing the fact that an Android library is not executable.
In a usual Linux environment BaseX stores all its database files inside a directory in the normal user directory. This means executing BaseX on Android without adjusting the directory to store the databases, would not work. BaseX receives the name and location of the home directory of the user by calling the Java function *System.getProperty("user.home")*, which works well with the JVM. Using this function on Android returns an empty string and results in BaseX trying to save the database in the root directory, what is impossible, because of the Android right system explained in Section 3.1.4. To avoid this behavior the above mentioned function needs to be replaced. The path where the data has to be stored must be a directory that only the application can access. Every application receives one directory with the given constraint from Android. This is where it will be installed as well as where the application will store its data. Additionally to this it is possible for an application to read or write on external storage, but this is not protected by the operating system

which means that every application is able access and modify those files.
The problem by using the private application directory is that BaseX should be a library project and it should be usable by every other Android project. So it is impossible to say how the directory is named and to locate where the database files have to be stored before even knowing how the application, which uses the BaseX library, is called. The explanation for this is that an Android application receives its directory by the name of its main package and not by the name of the library main package, so that more than one application can use this library. This is an issue that has to be dealt with, which is illustrated and discussed later in Section 3.3. Hence, it is necessary to tell the library this location and use it instead of its own package name. To implement this behavior a class has been created which is an entry point for the library. And it is the first instance when the BaseX Android library is used. This class creates an object of the library and needs the package name of the application as an argument for the method that returns an instance of its object. To create the object the Singleton pattern has been used [16]. The reason for this is that only one instance of the database can be created and used inside one application.
This class is called BaseXDatabase and creates a BaseX Context object by calling a constructor that has been added to the Context class. The newly created Context constructor checks if the given directory is available and if it is possible to read and write to this directory. Is it the first time the constructor of the class is used, it creates a directory which is called BaseXData. All databases created and used by this Android application are stored inside this directory. The directory is only accessible from this application[6]. At this moment it is possible to use BaseX as an Android library and it would create the needed BaseX directory.
To provide the BaseX operations the above mentioned class has been extended by additional methods which can be used for this purpose. All are implemented by using the above mentioned Context object, which means every operation is executed on this object.
After the migration process it is possible to use BaseX as it can be done on every usual PC which provides an implementation of the JVM. Except the cryptographic XQuery function, the BaseX Android library offers all functionalities like the normal BaseX desktop version offers. The Android principle that every application works in its own sandbox is also being kept. With this type of solution every application has its own BaseX database in its own directory. A possible advantage of this result is that the database does not need to consider queries that are coming from another application and synchronizations of the queries.
However, in the next section an alternative solution to this is provided by implementing the client/server mechanism of BaseX in Android. The advantages and disadvantages of the library implementation are outlined and investigated in Chapter 4. For the validation and performance measurement a test application of the BaseX Android library has been written and its execution performance analyzed, illustrated in Section 4.2.

[6]Assuming the application is executed on a non rooted device.

3.2.4 Providing a BaseX Android Client/Server Solution

At this moment every application which uses the BaseX Android library, has its own database which is able to create, read, update and delete the data. One disadvantage of that type of solution is that other applications are not able to use the data inside a database of another application. The reason for this, as mentioned in Section 3.1.4, is the sandbox principle of Android applications where applications can only access their own resources and not those from other applications. Speaking of a disadvantage could be misleading in this context, because this circumstance also prevents unauthorized applications to read or modify the data of another application. However, Android provides a mechanism which offers the possibility to provide data from one application to another. This technique is called Content Provider and is used for system wide databases like the database that contains the contacts of the owner of an Android device. Thinking of different applications that are using the same data shows that this could lead to redundant databases, which would increase the required storage size. In addition to this, every time the data changes on one point it has to be synchronized with every other database which stores the same data, too. For example, if every application has its own database storing the contacts and a new contact is added, all those databases need to updated.
Additionally to the stand alone version of BaseX it is possible to use it as a client/server solution. An instance of a BaseX client connects to a server and sends its requests and queries to it, which executes them and sends them back to the client. The advantage of this solution is that the client does not have to do the execution of the database related operations. Also the files which are containing the data, are stored and managed on the server and not on the client. Both shown advantages are reducing resource consumptions at the client instance. It also saves storage size as well as execution resources like the CPU or RAM occupation. But there is also a disadvantage, namely the client can only operate if there is a server available, and therefore it needs a connection to the server to send requests and receive responses.
As mentioned in Section 2.1 BaseX also provides a client/server solution, which also has been migrated to the Android platform. Hereby the focus does not lie on the execution of the server on an external device, like a server on the Internet for example. Although this is also possible and would result in a big saving of resources on the mobile device, but this would be similar to a web service which is not the focus of this thesis. The server client architecture of the Android BaseX version is to have one central database on one place and one instance of a server that handles all the upcoming database operations. In contrast to the, as outlined in the section before, Android library there is only one place on the device where the respective database files are being stored. If an application wants to use BaseX as a database it uses the client that sends the operations to the server, which is executed on the same device, and then receives the results. To achieve this Android provides Inter Process Communication (IPC) mechanisms which are offering the possibility to communicate between different applications.
However, the client/server version of BaseX uses the network and the TCP

protocol to do the communication between server and clients. This technique has also been used for the Android solution of the client and server implementation. Additionally the server has been created as an Android service. An Android service is a process which is executed in the background and is thus the last instance which is killed by Android process management. The reason to implement the server as an Android service is the process management of the operating system and the execution of the service in the background of the device. The constraint of limited RAM forces the Android process management to kill the not needed application and free their used spaces inside the RAM [2]. This can not be controlled, and the server has to be available all the time, so that an application can do its operation in exchange with it. The Android operating system tries to keep the service as long as possible alive, depending on different criteria and even it is being killed, for example by the user, it tries immediately to restart it. A service does not provide a graphical user interface and to communicate with it also IPCs are used. As for this an application has been written, to start and stop the server service. This application also has its own sandbox where the server stores its data. The application as well as the background service are executed in this sandbox. The data is stored in this sandbox, so no other application can access the data inside it, therefore the server is used. To achieve this goal BaseX uses, as well as the desktop version, the network for the communication with the clients. Specifically the client application does not need to implement IPCs, they send their requests via the network to the server and also receive the responses through the network. A graphical illustration of this principle is shown in Figure 3.7.

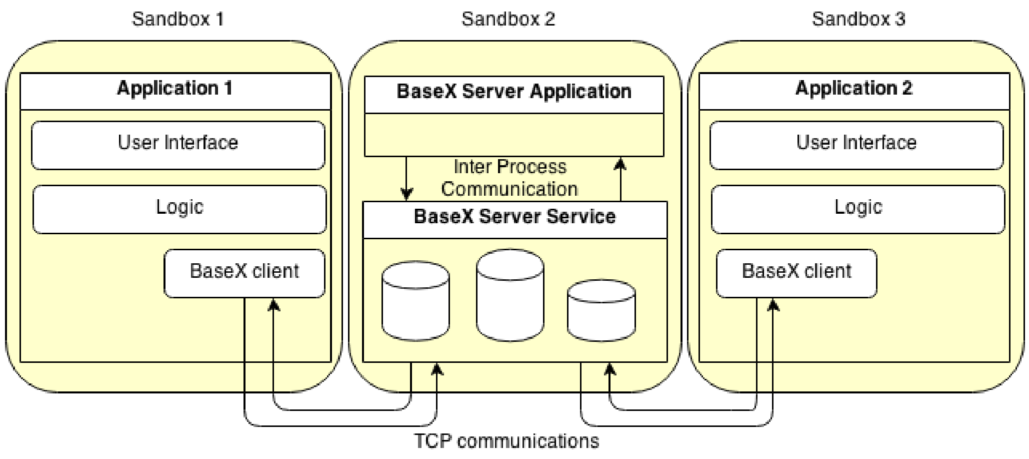


Figure 3.7: The principle of the BaseX server client communication in Android.

Every request and operation, which uses the BaseX client/server model, is executed on the device and thus there is no saving in execution resources, only in storage capability if different applications are using the same data. The implementation of the client server mechanism for Android has been done to illustrate that it is possible to have a server and client of BaseX running on one single device. The implication of this section is that it is also possible to have an external server which provides the data and handles the execution of any operation, which saves the above mentioned resources at the client. The only way to provide application data to another application is given when using the so called content providers supplied by Android itself. The BaseX client/server implementation could also be seen as an alternative for the present Android content providers.

Additionally the BaseX Android client as well as the library implementation are both implementing the same interface, so it is easy to replace one version with the other. This makes it easy to replace the library with the client solution if the resources have to be stored somewhere else, for example on an external server.

3.3 Problems During the Migration

During the migration of BaseX to the Android operating system several problems and issues occurred. The first problem which was discovered, was, as mentioned in Section 3.2.1, the lack of some included libraries, that are not supported by Android. To solve this issue they have been removed and their functionality has been deprecated. This has been done, because those specific functionalities are not required for a first runnable Android version of BaseX. This also applies for all graphical user interface components, which also have been removed, because they are not necessary for an Android library.
Another problem was the location of the BaseX database directory on the Android device. This need to be, as already described in Section 3.1.4, the */data/data/application-package-name* directory. The problem hereby is that one of the goals of the migration process is that the database is a library and not a standalone application. In the Android operating system the name of the private directory of an application depends on the name of its main package. This is only available if there is an Android application. A library does not offer this type of information, so it is not possible for a library to get the name or location of its private directory. The only way to get the package name of the application, except of hard coding it into the source code, is to get the application context and extract the name from there. This bears another problem, though only applications have a context, libraries do not, which also need to be considered. A possible way, could be to pass the context to the library, but then the library holds the context which it does not usually need and it will maybe not be freed from the garbage collector, due to the mark-and-sweep garbage collector strategy, as described in Section 3.1.1. Also the library would only need the application context to receive the application name and by using this information the name and location of the private directory. As mentioned above, holding an Android context massively slows down an Android application, so this is not the way how the problem should be solved. The alternative is to pass a string holding the application name to the constructor of the library, because

it only needs the name or the location of the *data/data* directory to access it. Therefore the context has been used before creating a library instance to receive the needed directory name, so that it can be passed right to the library through the constructor. The string, containing the directory name, then is used by the library to generate the needed directories. This can be done by the library, because it directly belongs to the application that uses it. As a result of this the library is executed in the same sandbox like the application and it has the corresponding rights. An issue with this solution is that the user directory which is used inside BaseX is a static string value initialized at the start of BaseX. All static values used by BaseX are initialized at start-up from within the static class Prop. And right after the start BaseX uses this value from this class in several places in the code. Which means that it need to be changed before it is being used anywhere in the code. For this reason a trick has been used to initialize the value before the constructor calls another constructor. In the BaseX *Context*[7] class the constructor which calls the actual constructor has been changed to initialize this value as a parameter. This results in the cycle of constructor calls, that the constructor which is called initializes the static values before calling the upcoming constructor in the chain. The corresponding Java code can be seen in Listing 3.1. The reason the file separator is not determined by using the available system method System.getProperty("line.separator") lies in the fact that the used separator is the same as in Linux. Namely it is the normal slash "/" and not a backslash as it is implemented inside the Windows operating systems.

```
public Context(String data_dir) {
  this(true, (Prop.HOME = data_dir + ‘‘/’’), (Prop.USERHOME =
      data_dir + ‘‘/’’));

  File dir = new File(Prop.HOME + ‘‘BaseXData’’);
  if(!dir.exists()) {
    if(!dir.mkdir()) {
     android.util.Log.i(‘‘BASEX’’, ‘‘CREATING BASEX DIRECTORIES’’);
    }
  }
}
```

Listing 3.1: The constructor of the BaseX context class.

Another issue which occurred during the migration process, was that BaseX intensively uses the Java string function isEmpty(), which is not available at an Android API-level lower than 9. The same applies for the copyOf() method in the java.util.Arrays Java package. This issue was solved by not supporting API-levels lower than 9 with the created BaseX library. According to the official Android homepage only 1.7% of the devices that are currently in circulation have an API-level lower than 9.[8] As a consequence developers can only create applications that are executable on Android devices using an API-level higher than 9, while using the BaseX library. To assure that no application, that is

[7]This is the *BaseX* Context class, and not the Context class, mentioned above, which is provided by Android.

[8]http://developer.android.com/about/dashboards/index.html

build on Android version lower than 9, uses the BaseX library the Manifest file of the library has been adjusted. This applies to one of the cases where the Manifest file of a library is used to determine the minimum Android version. Other constraints specified inside this Manifest file, like write permissions for example, are being ignored, as it was mentioned in Section 3.2.2.

Chapter 4

Analysis of the Ported BaseX Android Version

The present chapter outlines how the migrated BaseX Android database has been analyzed. The focus at this part concentrates on the evaluation and the performance of the library and its ability to be a possible database for mobile devices. First the specifications of the test devices are described and an estimation is made to show how the performance of the various devices differ. Therefore different benchmarks have been executed to approximate the distinctive hardware properties and their execution times. In the next chapter it is shown that the migrated database works without errors and failures on the Android mobile platform using a real device, while benchmarking the BaseX Android library. These benchmarks have also been used to identify the performance of the BaseX Android library. Therefore an XML benchmark set, which is especially designed to test the performance of XQuery executions, has been used. In aspects of performance, BaseX has been compared to the default Android database SQLite3 and a predication, which database system should be preferred for a specified use case, has been made. At the end of this chapter the constraints, in terms of maximal database sizes, of the Android version of BaseX used in the Android operating system has been inquired.

4.1 Evaluation of the Test Devices

For the measurement of the performance and the evaluation two different devices have been used. First, for benchmarking and evaluating the BaseX mobile version an Android Tablet PC Samsung Galaxy Tab 2 10.1 has been utilized. Second, for benchmarking the normal BaseX desktop version a Lenovo Thinkpad laptop has been used. Both not only vary in their operating system, they also have very different hardware specifications. Their important technical specifications can be seen in Table 4.1.

Device	CPU	RAM	filesystem	operating system
Laptop	Intel Core2 Duo L7100 1.2 GHz	2 Gb	ext4	Arch Linux (3.12.0)
Tablet	Dual-core Cortex-A9 1 GHz	1 Gb	ext4	Android (4.0.3)

Table 4.1: The technical specifications of the two devices, that were used for benchmark testing.

Looking at this table it can be seen that the only similarity of both systems is that they are using the same filesystem, which is ext4, which is short for *fourth extended filesystem.* The RAM of the laptop is the double amount of the tablet devices available RAM and both devices have a dual core CPU with around 1 GHz. Even if the specifications are not differ that much on the first look, their performance varies in some factors. Therefore a measurement of the input/output (I/O) and CPU speed of both devices has been made in Section 4.1 to get a clearer look how the devices differ.

The two devices which have been used to benchmark BaseX are totally different and not just in the fact that one is a laptop and the other a tablet PC. They differ in many hardware aspects, but there are two additional factors, that are interesting to identify a systems speed and are used for the given purpose. These are the CPU speed and the input/output (I/O) speed, where I/O speed is a headline for different operations. Broadly speaking it can be said that the I/O speed can be divided into read and write operations.
To measure these values the benchmark tool Bonnie++[1] is used. This tool executes different operations and measures how many of these operations can be executed in one second. It tests sequential output, sequential input, random seeks, sequential create and random create.
The sequential output represents the write speed of the system and Bonnie++ uses three different methods to measure this value. First it writes one character after another by using the `putc()` systemcall. After this operation it writes whole blocks with the size of 8192 bytes by using the `write()` systemcall and than calling the `close()` systemcall. The last test in this category is the rewrite test which differs from the write test in the fact, that the file is not closed after writing. Therefore one block write comply with 8192 `putc` calls and is way more effective and faster than writing single characters. As a next test the random seek operations per seconds were measured, which means how often the read/write position can be changed. It also measures the latency which represents the rotation speed of the disk, but this is obsolete because both systems using flash storage. Therefore there is no revolution per minute to measure.
Bonny is implemented in the C++ programming language and is available on most Linux distributions. But to use it on Android it is necessary to initially build it from its sources by using the, already mentioned in Chapter 3.2.2, Android Native Development Kit (NDK). This provides a cross compiler which

[1] `http://www.coker.com.au/Bonnie++/`

makes it possible to build C++ code for the Android platform. For the execution of the benchmarks Bonnie++ in the version 1.96 has been used. The results of these test executions can be seen in table 4.2.

	Sequential Output			Sequential Input		Random Seeks
	Char	Block	Rewrite	Char	Block	
Laptop	201K	62180K	23850K	907K	98239K	1161
Tablet	5K	20828K	8756K	596K	23768K	475.6
Factors	40.20	2.99	2.72	1.52	4.13	2.44

Table 4.2: Results of the Bonnie++ in- and output benchmarks.

Bonnie++ also measures the sequential and random create of a file. Therefore it creates a file, queries its status and deletes it by using the POSIX systemcalls `creat()`, `stat()` and `unlink()`. The results of this test can be seen in Table 4.3

	Sequential Create			Random Create		
	creat()	stat()	unlink()	creat()	stat()	unlink()
Laptop	17134	137752	13032	19442	170028	11774
Tablet	39	361	167	42	391	238
Factors	439.33	381.58	78.04	462.90	434.85	49.47

Table 4.3: Results of the Bonnie++ sequential/random create benchmarks.

By analyzing these values it can be said that the laptop is overall faster than the tablet PC. It is well known that I/O operations are the most expensive ones for use on Android, so the achieved result is no surprise. But with these values it can be said that there are factors which can tell how much faster it is in specific operations. This factors are impossible to optimize, because they are a hardware constraint and only changing the hardware can effectively improve them. But they are still important to identify the bottlenecks of the mobile BaseX version.
Analyzing Table 4.3 and having a look at the factors shows one value which is, compared to the others, extremely high. The sequential write per character is 40 times faster on the laptop than on the tablet PC. Considering this, it is clear to avoid writing by character instead by block. Writing by using the block mechanism is just three times slower than on the tablet device. The sequential reading of a single character is at the tablet just 1.52 times slower than at the notebook. Though the factor of the block reading is quite higher than the character reading factor. It is clear to say that block reading is much faster than character reading. But by using block reading the factor needs also to be considered in the BaseX benchmarks later in chapter 4.2.
Looking at Table 4.3 it can be seen that the sequential/random creation, reading and deleting is very slow on the Android device compared to the laptop. Considering this it should be avoided to often create files, because this type of

operation is very slow.
Bonnie++ gives a good overview about how fast the two system handle their I/O operations and how they differ in this type of aspect. But there is also another factor which affects the execution speed of a program, namely the CPU speed. It is obvious that this also differs on both test devices. To evaluate the CPU times a Java program has been written which executes four different CPU intensive operations. First it does the naive factorial of 5000, where naive means that it just iterates till 5000 and multiplies every step to the result. The second test is a recursive calculation of the 100th Fibonacci number. The third sorts an ascending ordered array of 10000 items using the bubble-sort algorithm. This is done because this is the worst case for the bubble-sort algorithm which has a complexity of $\mathcal{O}(n^2)$. The last test performs a naive test if the number 666667 is prime or not, by testing to divide the number by every possible candidate step by step till the candidate is the square root of the number. All these tests are very intensive in their CPU usage, because they only use arithmetic operations. The test have been implemented in the C programming language and have been compiled also using the Android NDK. The reason for this is that there is no distortion caused by the Dalvik or Java virtual machine. The results of these tests can be seen in Table 4.4.

	Factorial	Fibonacci	Bubble Sort	Prime Number
Laptop (avg. on 1000 runs)	98.19 ms	248.63 ms	180.53 ms	15.85 ms
Tablet (avg. on 1000 runs)	491.15 ms	2336.35 ms	1045.52 ms	76.41 ms
Factors	5	9.4	5.8	4.8

Table 4.4: Results of the CPU benchmarks.

Except the Fibonacci test it can be said that the CPU of the laptop is about five times faster than the one of the tablet device, respectively calculates and executes instructions five times faster. This is like the different I/O parameters a factor which could not be improved and has to be considered in the BaseX benchmarks as well. In general it can be said, that every operation, which is mainly CPU intensive, is executed five times faster on the laptop than on the tablet device. The same applies to I/O operations, depending on the type of the operation the laptop is up to 4 times faster than the tablet device. These evaluations are used for measuring the performance of the two BaseX versions and how to cope with the results achieved by two different platforms on two different devices.

4.2 Analyzing the Execution Performance

To investigate the performance of the Android version of BaseX the benchmark suite *XMark* has been used. *XMark* creates random XML files and provides a set of twenty predefined queries which can be used to measure the execution performance of an *XQuery* implementation. It offers the possibility to choose the size of the randomly created XML files, so that the test queries can be executed on different file sizes. Therefore it is also being used to search for the maximum supported database size of a BaseX database on Android in Section 4.6. All benchmark tests have been executed using the BaseX Android library and not the client/server solution. Additionally the Android benchmarks have all been executed inside the main thread of an application, which is usually not the case because all database operations should be handled inside a separate thread. The reason for this is to avoid unpredictable interrupts which could possibly hinder the benchmark thread. This is not the case if all benchmarks are executed inside the main thread.

4.2.1 The XMark Benchmark Suite

The XMark benchmark suite is used to identify the execution speed of the BaseX Android version. Therefore it offers 20 different XQuery queries which are especially designed to benchmark an XQuery implementation. The queries are separated into different categories, where every category targets a specific aspect of query execution. The XML files that can be generated by using XMark are all containing the same elements and every file is well-formed. The content of the generated XML files is randomly generated by using the 17000 most frequently used words[2] in the plays of Shakespeare [38]. The structure of the randomly generated XML files can be seen in Figure 4.1.

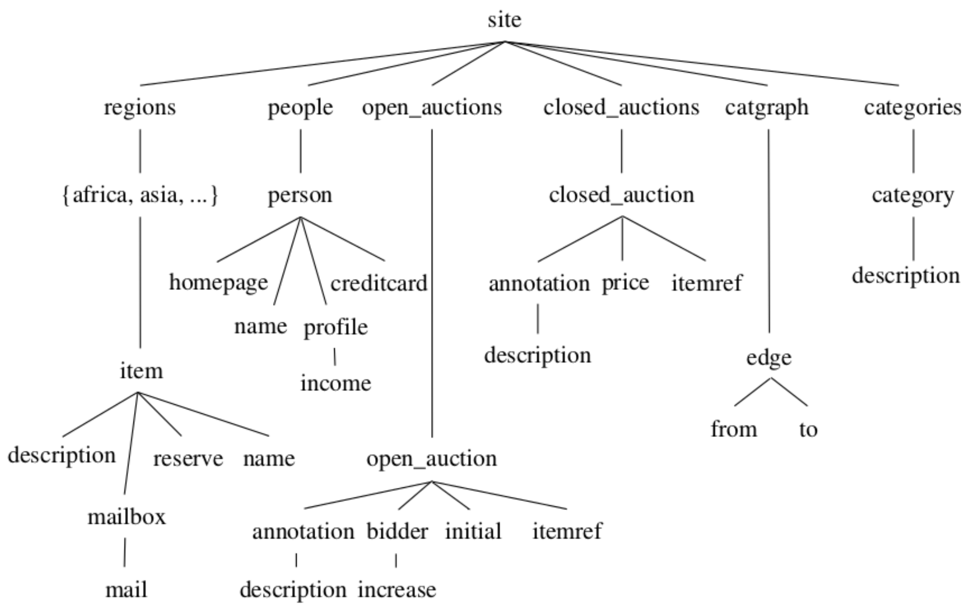


Figure 4.1: The structure of a generated XMark XML file. Source:[38]

[2]ignoring all stop words

The size of the XML files can be chosen during the generation process of XMark. For testing BaseX 15 different files have been generated. The smallest with a size of 100 KB, followed by files with a size increasing by 100 KB steps. Hence, the biggest generated file has a size of 1.4 MB.
The textual explanation of the different types of queries can be seen in Table 4.5.

1	Return the name of the person with ID 'person0'.
2	Return the initial increase of all open auctions.
3	Return the first and current increase of all open auctions whose current increase is at least twice as high as the initial increase.
4	List the reserves of those open auctions where a certain person issued a bid before another person.
5	How many sold items cost more than 40.
6	How many items are listed on all continents?
7	How many pieces of prose are in our database?
8	List the names of persons and the number of items they bought. (Joins person, closed_auction)
9	List the names of persons and the names of items they bought in Europe. (Joins person_auction, item)
10	List all persons according to their interest; use French markup in the result.
11	For each person, list the number of items currently on sale whose price does not exceed 0.02% of the person's income.
12	For each richer-than-average person, list the number of items currently on sale whose price does not exceed 0.02% of the person's income.
13	List the names of items registered in Australia along with their description.
14	Return the names of all items whose description contains the word 'gold'.
15	Print the keywords in emphasis in annotations of closed auctions.
16	Return the IDs of those auctions that have one or more keywords in emphasis.
17	Which persons don't have a homepage?
18	Convert the currency of the reserve of all open auctions to another currency.
19	Give an alphabetically ordered list of all items along with their location.
20	Group customers by their income and output the cardinality of each group.

Table 4.5: The XMark queries. Source:[38]

The queries are divided into various categories, which are aiming to benchmark several functionalities and concepts of XML. Category one includes just Query 1 and tests the performance of an exact match. The second category analyzes the behavior of the database by querying it with order constraints. This category contains the queries 2, 3 and 4. Casting is the purpose of the third category, which only includes Query 5. The next category contains the queries 6 and 7, and tests the regular path expressions. Chasing references is the topic of the fifth category that contains Query 8 and 9. Constructing new elements and querying them is the purpose of the next category that includes only Query 10. Benchmarking the execution of queries with a large result set by using joins is the goal of the seventh category, which involves Query 11 and 12. Query 13 is the only query in the next category, which aims to test the performance of reconstruction of a document. Search a full text by using a key word is the purpose of category 9 that just includes Query 14. The next category tests the performance of deep path traversals without wildcards, this includes the Queries 15 and 16. Category 11 includes Query 17 and investigates the performance of the database by querying missing elements. User defined functions are the content of the next category which only contains Query 18. To investigate the performance of the database executing a query that sorts the result is the purpose of category 13 which is achieved by Query 19. The last category is used to test the speed of an execution of a simple aggregation by using the last query [38].

4.2.2 The Results of the Benchmark Execution

To execute the XMark queries two applications have been developed, one for testing the BaseX desktop version and one for the Android version. These two programs are pretty much the same except for the target platform and the used BaseX version. Both feature the same functionalities while they operate equally. At the first step the programs create 15 databases and add the 15 XML files, that were mentioned in the section before, to these databases. After this operations the application opens one database after another and executes every XMark query on it. Every query is executed a hundred times and the average time consumption is being calculated and stored into a specific file. The result of the execution of the application using the laptop can be seen in Figure 4.2 and the result of the execution using the tablet is shown in Figure 4.3.

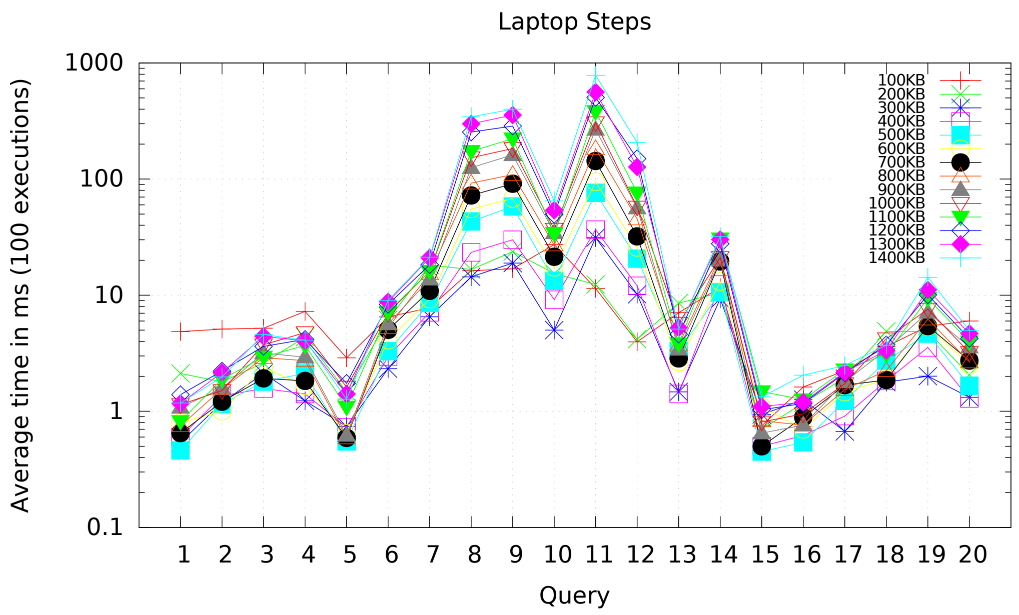


Figure 4.2: The results of the XMark benchmark queries executed on the laptop device.

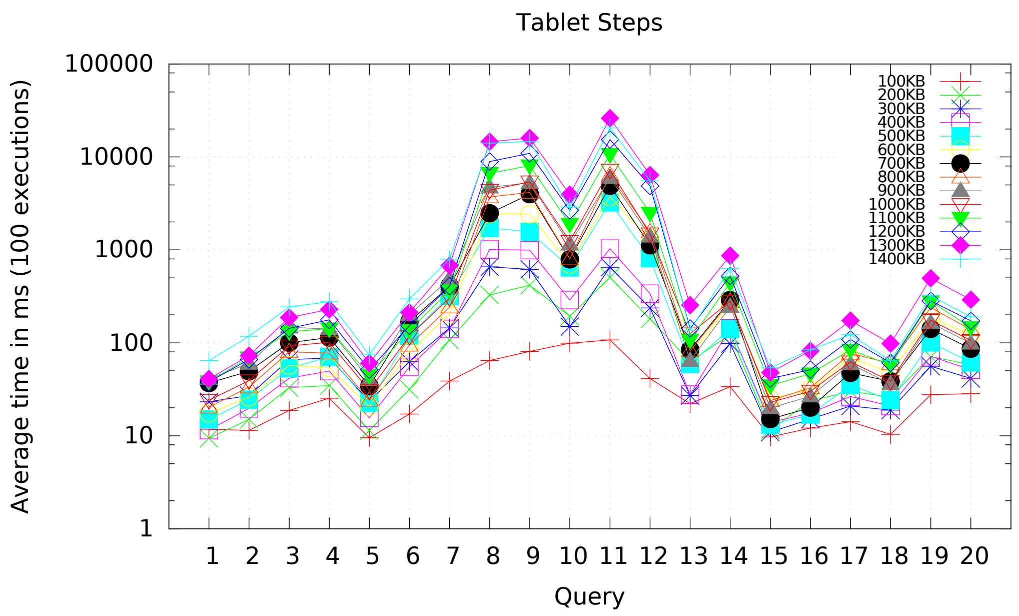


Figure 4.3: The results of the XMark benchmark queries executed on the tablet device.

Looking at both images it can be seen that with an increasing size of an addressed database also the execution time of the performed test queries increases. This behaves as expected, because all queries have a complexity of at least $\mathcal{O}(n)$ and with an increase of the file size the amount of elements are also increasing. It is also shown that the curves of both images are similar to each other, which is a sign for the fact that no unpredictable circumstances have occurred. A query which is very fast on one device and the slowest on the other or vice versa would be an example for this.
Although the two graphics are looking very much alike they differ in one important aspect, the fact that the execution time is up to seventy times higher using the tablet PC instead of the laptop device. Even if this factor is just the highest one, the queries which are executed using the tablet are around thirty times slower, on average, than the ones executed on the laptop.[3] This results are giving a first overview of the performance of BaseX running on the Android platform. The perception that BaseX is faster executed on the Laptop was anticipated, by just considering the lack of hardware resources at the Tablet shown in Section 4.1. Remembering the fact that, for example, the CPU speed is about five times faster on the laptop than on the tablet PC it still needs some investigation on how such factors, like the seventy times higher execution time, are being achieved. Another interesting insight is the fact that the execution time is sometimes faster using a bigger database than a small one. This only affects the benchmarks made using the tablet device. In general it cannot be said how the circumstances mentioned above actually occur. Hence, more research needs to be done with the Android version of BaseX, which is shown in the next section.

4.2.3 Identifying the Bottlenecks

With the Android SDK various development tools are provided. On of them is Traceview, which offers the possibility to record a specific part as source code of an execution of an application. Traceview provides a graphical view to analyze this records or the possibility to transform them into HTML code. The content of such records implies every method call and its execution time, as well as the occupation of the CPU in percent and the amount of calls. The execution times are given in microseconds, which are not representing the real world time, the value represents absolute CPU occupation time. This fact makes the recorded trace-views very valuable, because no interrupts are tampering the results.
All twenty XMark queries have been recorded using Traceview by executing them with a database that features the size of one mega byte in total. Most records contain a very long list of method calls, even if they only record a small part of the code. Therefore only the top five time consuming methods for every query have been investigated. Summing up those methods it can be said that there is an amount of twenty methods which apply to the group of the top five time consuming methods that have been recorded using Traceview. Traceview

[3]The complete results of all executed XMark benchmark tests of the laptop and the tablet device are shown in Appendix A. A specific table is also given which includes the factors that show how the various execution times differ from each other.

also records the amount of calls that every method experienced. With this additional information it is possible to calculate the average time spend inside one method. Table 4.6 shows the five methods which feature the highest values as a result of this calculation.

$\frac{cycle}{calls}$	Name
32592	dalvik/system/VMDebug.startGC
1121	util/Compress.unpack
277	value/node/DBNode.uri
175	query/path/IterStep$1.next
105	util/Token.norm

Table 4.6: The five methods with the highest $\frac{cycle}{calls}$ value.

The most time consuming method is, as shown in Table 4.6, VMDebug.startGC. This function is called only once and executes a long time in average. According to [40] this method is a fake method. It is implemented to display the execution of the garbage collector of the Dalvik virtual machine on the Traceview records. Unlike most implementations of the Java virtual machine, it is not possible for the Dalvik VM to change the garbage collector strategy. It is possible to increase the size of the process's own heap, which indirectly affects the garbage collection, by slowing it down with a bigger heap size. The available heap is hereby device dependent and increasing the size can cause crashes of the application, with out of memory exceptions. More about the specific heap size of an application, executed on the Tablet, is described in Section 4.2.4. Since Android version 2.3, which is the minimum version for the use with BaseX Android library, the garbage collector is implemented in a concurrent way and does not influence the executing thread [12]. This is also the reason why it is executed only once and its execution takes so long. The GC thread is started for one time and collects all unreferenced objects till the process is finished. Therefore this method is being ignored, because increasing the heap size is not an option and the garbage collector is not executed within the thread which benchmarks the BaseX code.

Looking at those methods, the DBNode.uri method is the only one of those, which provides a possible way to be optimized. This is done by modifying the corresponding Java source code. Unfortunately, this method is not the top time consuming method. Compress.unpack is the slowest executed method, that was recorded using Traceview. It consumes an average of 1121 cycles per call, which is very slow compared to the other methods. The purpose of this method, inside BaseX, is to decompress the given byte array. During its execution it iterates over the given byte array and decompresses all characters.
The IterStep$1.next method is part of an iterator and each time it iterates to the next node. Depending on the amount of nodes, the time spend in this method and the corresponding calls to it can in- or decrease. This results in the conclusion that this method could also not be optimized for the Android operating system.
Token.norm is used to normalize all whitespaces in the byte array which is passed

using the parameter of this method. By normalizing it is meant to search for horizontal tabs, line feeds, carriage returns and spaces inside the byte array and then replaces them by an empty character. This is done in a normal for-loop and could also not be improved by changing it. The only method, that has the potential to be optimized by modifying the corresponding Java source code is the DBnode.uri method, which is shown in the following section.

Analyzing the DBNode.uri Method

The third most time consuming method is the DBNode.uri method with an average of 277 cycles per call. Listing 4.1 shows the corresponding source code for the NSGlobal.uri method.

```java
public static byte[] uri(final byte[] pref) {
  for(int s = NS.size() - 1; s >= 0; s--) {
    if(eq(NS.name(s), pref)) return NS.value(s);
  }
  return null;
}
```

Listing 4.1: The code for the uri method in the NSGlobal class.

Looking at the source code for the uri method shows that it only executes a for-loop and checks if the parameter pref is equal to the NS.name byte and if this is the case the value is returned. The method has a complexity of $\mathcal{O}(n)$ which results in the assumption that the for-loop is always executed in the worst case. This would mean that every time the method is called the loop is fully iterated from the maximum size of the NS object till zero. Considering the, in Section 3.1.1 mentioned, Just In Time compiler from the Dalvik VM this part of code should be optimized by it. Even if this part of the BaseX Android source code is compiled into native machine code it is still very slow, compared to the other expensive methods. Besides the for-loop there are the eq and NS.name methods, which also could produce the long execution time of this function. The eq method compares the two committed byte arrays for equality, implemented by using a for-loop which iterates over the arrays and compares their bytes. This for-loop is executed very often and there are no other method calls inside of it, so it is very obvious that this loop is compiled into machine code by the JIT optimization routine.
The investigation of the NS.name method shows that this function is just a simple getter method which returns the name from the index of the parameter. At this place the differences between the two platforms are playing an important part, because it is a best practice to use getter and setter for the normal Java environment, but it is expensive for the Android platform [42]. Even if the JIT compiler inlines the getter/setter calls it can be up to 30% faster if a direct field access is used instead of the getter/setter methods [41].
In Section 4.4 the getter of the *NSGlobal.uri* method has been replaced by direct field accesses and it is shown that it improves the execution time of Query 2 for the BaseX Android library. This has also been applied for the BaseX desktop version and investigated if there is also an improvement of the execution speed.

4.2.4 Analyzing the Hardware Usage

In Section 4.1 it has been shown, that the laptop computes five times faster, in average, as the tablet PC. Additionally to this the write operation on the file system is up to three times slower on the tablet compared to the laptop and the read operation is about four times faster while performed on the Laptop. Thinking about those factors could lead to the solution that most of the longer execution times on the tablet device are achieved by the given hardware constraints. The created Traceviews also show a often use of the TableDiskAccess class. This class offers methods to store the data on the disk or to read it, therefore it uses block-wise operations. The measured values of Section 4.2.4 for the I/O operations on the test devices, illustrates that writing is three times faster on the laptop than on the tablet PC. And reading is even more than four times faster compared to the Tablet device, by using block wise operations. For the XMark benchmark test a write operation has not been significantly executed, because the benchmarks only query the data and does not create new one which is stored permanently in the database. Therefore no Traceview shows the usage of the writeBlock method for example, the analyzing of the write performance of the BaseX Android library is shown later in this section.
In contrast to the write operation, the read operation is used very often. Even if it is not part of the most time consuming methods in the Traceviews it is still the factor of four which always needs to be considered, when it is being executed.
To measure the performance of the write operation an Android application has been created which creates databases and fills them with different data. The provided data has been created using the given XMark tool to create random XML files. Therefore files with a size of 1, 5 and 10 MB have been generated. As well as testing the Android version of BaseX, in its write ability, the desktop version has also been tested with the same set of data. The results can be seen in Table 4.7.

File size	Laptop	Tablet	Factor
1 MB	168.54 ms	3389.55 ms	20.11
5 MB	737.88 ms	17418.61 ms	23.60
10 MB	1543.37 ms	36246.68 ms	23.48

Table 4.7: The average times of the create database operation, for 50 runs.

The results of the measured write operations illustrate that the laptop in average is about twenty times faster in writing operations than the tablet device. To narrow it down, why this big factor occurs while creating a database on the tablet PC, the Traceviews for the create database operation have also been recorded. The Traceviews have been recorded using all three file sizes, but they all have the same ordering of top time consuming methods and they differ only in their execution times and calls. Therefore the 1 MB example has been used for analyzing the recorded Traceview.
Looking at the top five most time consuming methods inside the Traceview of the create operations shows that all of them are read and not write operations.

The reason for this is that BaseX spends the most of the time inside a create process to parse a given XML file, due to flat table representation [20].

$\frac{cycle}{calls}$	Name
5.69	XMLInput.read
5.46	XMLScanner.consume
5.70	TextInput.read
5.67	NewlineInput.read
5.50	TextDecoder$UTF8.read

Table 4.8: The five methods with their $\frac{cycle}{calls}$ value for the create operation.

Analyzing the top time consuming methods, gives an average $\frac{cycle}{calls}$ of a factor of around five, which is nearly the same for every method, as shown in Table 4.8. This also applies for the Traceviews of the create operation of the 5 and 10 MB file size databases, where their number of cycles increase as well as their amount of calls. Also analyzing the code of those methods does not provide any possibility to effectively improve their performance.

The Android operating system has a process management which is optimized for mobile devices with low resources. Android provides two different types of available RAM. First the private one, which is only available for the application itself, and secondly the shared RAM which is split between the running applications. The process management suspends processes which are not being executed to the background and if there is a lack of available RAM it starts to kill those processes and free their memory till enough RAM is available again. This can not be affected by the application itself. Therefore it cannot be measured how much RAM is in use by the actual executed application and which is occupied by other processes that operate in the background. The same applies to the available CPU usage, depending on the decisions of the operating system the application receives CPU time or not. These constraints are also playing a role in classifying the performance of the BaseX Android library, because those are factors which cannot be influenced directly.
The Android Development Kit provides its application developer a tool that offers the possibility to view log messages during the execution of an application. This tool is called Logcat and can be used in combination with the Android Debug Bridge. The used garbage collector inside the Dalvik virtual machine always prints a message to the Logcat when it starts to free memory. This information can be used to receive an overview on how often the garbage collector is triggered and how much memory it frees. The information displayed are the reason for the garbage collection, the freed amount of memory, the heap stats and the time needed for the garbage collection by itself.
Depending on the available heap size of the process this output varies for every execution, due to the memory management mechanism of Android. However, the Logcat output of the garbage collector for Query 5 and Query 11 has been recorded. Query 5 is the query with the least amount of time consumption and Query 11 is in contrast to this the most time consuming query. This fact is

also reflected in the corresponding output of the Logcat information. Namely the garbage collector is called only one time by the execution of Query 5 on a 1 MB database and freed an amount of 540 KB in total, pausing the execution for 6 ms. In contrast to this minimal usage of the GC the messages shown by the Logcat during the execution of Query 11 are showing an amount of 25 of GC usage. The values summed up can be seen in Table 4.9.

Query	GC executions	Freed memory	Time used
5	1	540 KB	6 ms
11	25	15173 KB	226 ms

Table 4.9: Garbage collection statistics during the execution of Query 5 and 11.

This illustrates that the garbage collector of the DVM also slows down the execution of BaseX on the Android platform. The last column of the table displays hereby the real time which the application is paused due to garbage collection. The garbage collector is executed within its own thread which is scheduled by the Android operating system and needs an average time consumption of 40 ms for every execution.
Usually the available RAM for every process is between 16 and 128 MB. At the used tablet device an executed application receives 48 MB of RAM for its execution, which compared to typical desktop systems is very low. However, there is a way to increase this size manually for applications which are needing a bigger RAM size. This can be done in the Android Manifest file by adding *android:largeHeap="true"* and resulting in a total of 256 MB RAM available for the BaseX application on the Tablet test device. The increase to a maximum of 256 MB of available heap size does not mean that the application has this amount as default available. Actually it means that the application can request more heap if it is running low on it and the operating system will grant it until the maximum capacity of 256 MB is reached. Additionally this option is only used for testing purpose, because it is just available for Android versions higher or equal than API level 11 and therefore cannot be written from within the Manifest file of the BaseX Android library.
The result of the test is that the artificial increasing of the heap size has not effected the garbage collection of BaseX Android library, because only small objects are being allocated actually. The purpose of the option to increase the heap size is to load large objects into the heap, which under normal circumstances would run out of memory, for example with the use of large bitmaps. Although BaseX allocates a lot of memory, but it does not keep it, the garbage collector frees it due to the fact that the allocated objects are not be used any more. This also is a reason for the frequent occurrence of the garbage collector while executing Query 11 from the XMark benchmark suite. And this is another factor which slows down the BaseX Android version because of the restrictions given by the operating system and the virtual machine, which cannot be avoided.

4.3 Differences of the Mobile and Desktop Version of BaseX

Despite the listed differences in Sections 3.2.3 and 3.3 there are more distinctions between both versions. The Java Development Kit (JDK) offers a tool which is called Hprof. This is similar to the Traceview tool provided by the Android SDK and offers the possibility to record the execution of a specific class or method. The Android tool Traceview is actually an extension of Hprof that expands Hprof with some functionalities. Unfortunately some of those functionalities have been used to identify the most time consuming methods at the BaseX Android version and therefore those are missing at Hprof. This missing functionalities and the fact that Hprof is not accurate makes it not as valuable as Traceview [30]. However, Hprof provides the ability to be used to identify methods which are being executed the most. It does not provide the CPU occupation time in cycles, like Traceview, but it displays it in overall percent and it also offers a list of the heap allocations made by the recorded program [26]. This information can be used to identify the top time consuming methods, which can then be compared with the most top time consuming methods of the BaseX Android version. The time spend in those methods and the amount of calls to those methods are not displayed, but it can be said that those methods are being compiled into native machine code by the HotSpot Just In Time compiler of the Java Virtual Machine. As explained in Section 3.1.1, the JVM uses a method based JIT, that compiles the so called hot spot methods, which are the ones displayed by Hprof.

Analyzing all Hprof dumps illustrates the additional differences between the two BaseX versions. None of the top five time consuming methods of the recorded Traceviews from Section 4.2.3 are actually represented inside the Hprof records. Remembering the fact that the VMDebug.startGC method is part of the Dalvik virtual machine, which is responsible that it is not represented in any record of the Hprof runs. The second most time consuming method of the benchmarks of the Android version of BaseX is the Compress.unpack method, as already shown in Section 4.2.3. This method is also represented as one of the top time consumers inside the Hprof records for the desktop version. The DBNode.uri method was the third most time consuming method in the Android version before its improvement. At the desktop version of BaseX it is also listed in the recorded Hprof executions, but it is not one of the slowest methods. This also underlines the differences between the mobile and the desktop version, because at the not improved Android version this method marks one of the existing bottlenecks. As mentioned in Section 4.2.3 the Just In Time compiler of the Dalvik virtual machine could be a reason for such a behavior. In contrast to this the JIT of the Java virtual machine seems to translate the getter calls to native machine code, which can be an explanation why the DBNode.uri method is not in the top time consuming methods of the Hprof records. Except for this difference no other significant differences in the Traceviews of the BaseX Android and the desktop version have been extinguished. Additionally to this there is a small distinction between the two platforms, which is not listed in

the most time consuming method. There are several input/output methods in the Traceviews of the Android version which are not listed that high inside the Hprof records. A possible explanation for this is the faster I/O operation speed of the hardware of the laptop device compared to the hardware of the tablet PC, as shown in Section 4.1.
The analysis and comparison of the Hprof records and the Android Traceviews are underlining the assumption that most of the bad execution performance achieved by the BaseX Android version is caused by the lack of hardware resources on the given mobile device.

4.4 Results of the Improvement

One aspect which was found as a possible improvement, by analyzing the execution times and the corresponding Traceviews from Section 4.2.3, is the getter and setter part. As mentioned in Section 4.2.3 the use of direct field access instead of getter/setter methods improves the execution time by 30%. This value is shown by Tonini[42] and is the change which has been done for the BaseX Android library. The recorded Traceviews have shown that especially the DBNode.uri method features a high execution time caused by the getter calls inside the for-loop. This call accesses the nm byte array which is one part of tuples of name and value pairs in the container class Atts. All calls to the getter and setter to these tuples have been replaced by direct field accesses inside the whole BaseX Android library.
The execution of the XMark benchmark tests with this optimized version of BaseX on the tablet PC showed an improvement of up to 200%, which is nearly three times faster as the normal version of BaseX. Especially the time consumption for the execution of the queries 2, 10 and 17 have been improved. An extensive use of the DBNode.uri method with those queries is responsible for this behavior. The average improvement of this change is about 58.6%. Analyzing the Traceviews of the improved version shows, that the average execution time of the DBNode.uri method has been nearly cut into half and it is not even close to the list of the top time consuming methods. The results of the whole execution can be found in Appendix A.4 and the improvements for every query in percent is illustrated inside Table A.5, which is also available in the Appendices chapter at the end of the present thesis.
The replacements of the getter and setter calls at the desktop version of BaseX have no significant changes in the time consumption of the overall execution time. This was also tested on the laptop device, which underlines the statement that this mechanism, to speed up an application, is only working at the Android version and is the result of the used JIT from the Dalvik virtual machine. The average improvement of 58.6% is even higher than the 30% mentioned before and achieved by Tonini[42]. This shows, that the replacement of the getters/setters is a technique which need to be continued in the BaseX library project, as well as in every other time consuming Android application. Responsible for the improvement on the Android platform is the, in Section 3.1.1 and before mentioned, Just In Time compiler of the Dalvik virtual machine. Even if the

compiler copies the getter and setter methods to the corresponding place inside the code, the JIT is not able to transform them into native code, therefore they have to be replaced with direct field accesses. Using direct field access enables the JIT to translate those accesses into native machine code and this is responsible for the boost of time improvement. Figure 4.4 shows that the curves still have the same shape, which indicates that the changes had an impact on the whole execution of the benchmark suite. The most time consuming queries inside the benchmark tests of the unimproved version are still the most time consuming methods in the improved version and the same applies for the fast queries. The Queries 8, 9 and 11 are still the slowest ones, but the overall time consumption could has been lowered instead.

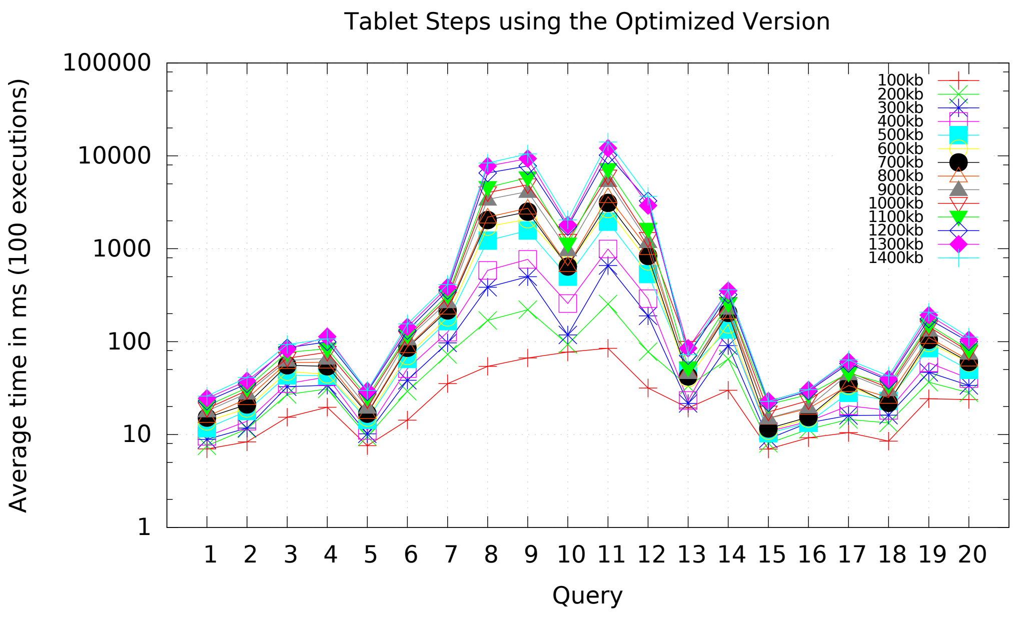
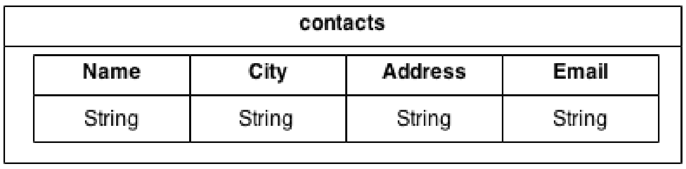


Figure 4.4: The results of the XMark benchmark queries executed on the tablet PC, using the optimized BaseX version.

Compared to the old execution time, the adjustment results in an enormous improvement. However, compared to the times achieved on the laptop devices the benchmarks are still slow on the tablet device. It is hard to tell where the most time of the execution is spend and how this affects the performance of BaseX on the Android operating system. In general the Android virtual machine Dalvik is designed to be a fast VM that executes its code in less instructions than an implementation of the JVM. The implication of this should be a faster execution of the BaseX code on the Dalvik VM than on the JVM. The lesser hardware capabilities of the Tablet are playing a big role in the execution of the benchmarks, especially the available RAM for the DVM, which has been shown in the section before. Therefore the best improvement would be more available hardware resources in order to speed up BaseX significantly.

4.5 Comparison between BaseX and SQLite3

Even after an optimization of the BaseX Android version the execution of the XMark benchmark tests are still significantly slower than the desktop version. It has been shown that the main part, which is responsible for this behavior, is the limited hardware resources available on mobile devices and some restrictions in use with the Dalvik virtual machine have been extinguished. Therefore the execution times of BaseX are now being compared to the execution times of the SQLite3 database, which is part of Android, as already described in Section 3.1.4. SQLite3 is a relational database which uses SQL as its query language and not XQuery, which is used by BaseX instead. Thus, to benchmark both database systems and compare them to each other the queries have to be very similar as well as the used data inside the databases. The used data for the comparison of the two databases are identical and consists of a set of contacts, including the name, city, post- and email address of a person. In the SQLite3 database all this columns are represented as string values and all have to be filled, the same applies for the database inside BaseX. The structure of the respective SQLite3 table can be seen in Figure 4.5. The corresponding XML structure used in BaseX in Listing 4.2.



contacts

Name	City	Address	Email
String	String	String	String

Figure 4.5: The structure of the table used inside SQLite3.

```
<contact>
  <name>Name</name>
  <city>City</city>
  <address>Address</address>
  <email>Email</email>
</contact>
```

Listing 4.2: The XML structure of the contacts database for use with BaseX.

Depending on the amount of contacts eight different sizes of databases have been used for the benchmark test. Starting from small to big the databases are containing 10, 50, 100, 500, 1000, 1500, 2500 and 5000 contacts. There are six different statements which have been used for the benchmark purpose. The SQL statements as well as their corresponding XQuery code can be seen in Appendix B.1. They are divided into two categories; one is querying and the other is modifying the given data. During the execution of the statements the time of their execution was measured and their exact functionality can be seen in the following list:

1. Get all contacts by their name in ascending order.
2. Get all contacts having an email address ending with “com”.
3. Search for a specific name.
4. Delete every contact having an email address ending with “org”.
5. Insert a new contact.
6. Update the recent inserted contact.

The statements listed above are not intensive statements which would be used to stress test the database, as it has been done in Section 4.2 with the XMark benchmark queries. In contrast to this the statements are easier queries which could also occur in the use case of a real application. The benchmarks have been executed using the tablet device and the optimized BaseX Android library. The consumed time of the query statements and the different databases can be seen in Table 4.10, as well as the results achieved by the modifying statements in Table 4.11.

	Statement 1		Statement 2		Statement 3	
Contacts	SQLite3	BaseX	SQLite3	BaseX	SQLite3	BaseX
10	2.00	23.77	1.70	24.10	1.15	24.38
50	12.14	50.81	2.44	50.57	1.38	41.74
100	13.67	91.43	3.58	58.56	1.52	56.12
500	28.93	149.96	9.61	100.12	2.62	276.09
1000	76.41	269.65	14.92	180.38	4.05	353.94
1500	79.41	401.67	25.90	308.38	5.79	435.97
2500	198.57	752.74	51.66	501.06	9.73	684.26
5000	419.86	1491.21	55.41	992.18	20.05	1366.02

Table 4.10: Measured execution times for the query statements in milliseconds.

	Statement 4		Statement 5		Statement 6	
Contacts	SQLite3	BaseX	SQLite3	BaseX	SQLite3	BaseX
10	4.39	9.67	32.65	9.85	4.18	10.16
50	4.91	21.11	28.32	11.01	4.27	14.22
100	28.77	22.76	32.44	9.67	4.33	32.19
500	355.49	88.19	52.67	10.32	5.43	80.96
1000	728.27	178.00	41.17	10.23	5.92	111.81
1500	972.47	311.06	72.33	10.22	6.43	192.90
2500	1357.69	570.00	152.85	10.28	7.29	253.44
5000	2270.29	1016.99	364.71	15.96	10.65	594.90

Table 4.11: Measured execution times for the modify statements in milliseconds.

Looking at the results of the query statements in Table 4.10 illustrates the speed advantage of SQLite3 compared to BaseX. On both database systems

the time consumption depends on the amount of used data and there is nearly a linear growth in every query on both databases. Overall, the general result is that SQLite3 is faster in every query statement than BaseX. In query 1 it is between three and four times faster, in query 2 it is around 20 times faster and executing statement 3 is around 60 times faster using SQLite3.
In contrast to the query results the values of Table 4.11 are showing that inserting and deleting of data is both faster with BaseX than with the use of SQLite3. Deleting all data with an *org* ending in their email is nearly twice as fast using BaseX instead of SQLite3. Inserting a new contact into the database is always performed in a constant time using BaseX, while it increases with the size of the database using SQLite3.
The fast query times are one advantage of SQLite3, the reason why this is achieved is that SQLite3 is specially implemented for mobile devices. Similar to the BaseX Android version, SQLite3 does not use a client/server architecture and therefore every application has its own database[43]. SQLite3 is a system library of Android, it is implemented in the C programming language and therefore needs no virtual machine for its execution, which makes it faster in the aspect of time consumption. Another reason for the fast execution of every query is the use of a buffer cache, which is used by SQLite3 and cannot be disabled to measure the absolute performance of the database [22]. Making a query in SQLite3 on an Android device also returns a cursor object pointing to the first result and not the whole result. To receive the whole result, as a string for example, the received cursor is used to iterate through the results. For this comparison the same has been done with BaseX, an iterator object is the result of a query and not as usual a string containing the results. This saves the time and resources which would be used to build the result as one single string.

In the XMark benchmark tests the whole results, received by a query, have been whole strings. This fact is also partial responsible for the achieved execution times. As mentioned before, to get a result as a string the received cursor of a SQLite3 query has to be used to iterate over the result set. To get an overview how BaseX and SQLite3 handle the queries by returning the whole result as a string this has also been measured and the results can be seen in Table 4.12.

The results displayed in Table 4.12 illustrate, depending on the size of the result, that BaseX achieves a still tolerable time with 1.6 seconds by querying 5000 datasets and receiving it as a string. In contrast to this SQLite3 is during the execution of the same operation around 200 times slower, which is caused by the part of building the string by iterating over the returned result set. The table also displays the correlation between the required time and the size of the string. This applies to both SQLite3 as well as BaseX. Query 1 returns all contacts, while query 3 only returns one record and therefore is faster, because of a smaller result iterator and the consequent result string. Even if query 3 just returns one record and query 1 5000, the time consumption of both queries using BaseX are differing by just 300 milliseconds. The same applies for the smaller databases, BaseX processes the queries in nearly the same time. For the described use case, this constitutes an advantage of BaseX over SQLite3,

	Statement 1		Statement 2		Statement 3	
Contacts	SQLite3	BaseX	SQLite3	BaseX	SQLite3	BaseX
10	3.90	45.10	1.31	56.94	1.12	25.32
50	41.44	49.22	2.13	27.40	1.37	35.49
100	108.33	108.85	4.63	52.52	1.52	50.47
500	1891.66	203.27	75.19	95.61	2.99	247.65
1000	7996.00	272.09	383.75	219.11	4.05	306.33
1500	17040.43	426.26	886.87	355.22	5.58	450.59
2500	56181.91	831.93	2540.80	566.83	8.51	703.73
5000	211718.96	1684.96	9407.74	1090.27	15.59	1316.77

Table 4.12: Measured execution times for the query statements returning the result as a string in milliseconds.

because there is no worst or best case of the queries. The execution time just depends on the size of the database and not, as it is in SQLite3, on the size of the received results.

Those results are illustrating that applications that consist of many queries should use SQLite3 and otherwise applications with an huge insert and delete ratio should prefer BaseX as their database system. At this conclusion it need to be considered how the results of a query should be handled, if they are iterated later or if they should being displayed after receiving them from a query as a single string, for example. For the last circumstance BaseX should be preferred, because querying and receiving the whole result as a string is still fast compared to SQLite3.

Looking at some results of all tables in the given section shows that there are sometimes variations in the linearity of the results. An example for this is the insertion of a contact in the database with the size of 50 contacts using BaseX, as shown in Table 4.11. There can be more than one possible clarifications for this circumstance. One could be the execution of the garbage collector at the given moment, or another interrupt of the operating system has been executed. The Android process management, as described in Section 3.1.4, makes it impossible to avoid such interrupts and therefore those erratic fluctuations.

Unfortunately it was not possible to compare BaseX with the XQuery processor MXQuery, mentioned in Section 2.3, which should also be available for the Android operating system. The reason for this is that it seems that MXQuery will not be further developed since its initial release back in 2011. The Android version of MXQuery uses an old library of Apache Xerces[4] as its XML parser, which is deprecated and not supported by Android any more. Therefore it could not have been compared to the BaseX Android version, and makes BaseX the only available XML database and XQuery processor for Android, at the moment.

[4]http://xerces.apache.org

4.6 Defining the Constraints of the BaseX Android Version

On mobile devices there are not as many hardware resources available as on desktop computers. Therefore in this section the constraints have been searched, which are representing the maximal boundaries for using the BaseX database in a mobile environment.
However, the great variety of Android devices makes it nearly impossible to make a general statement about the constraints of BaseX, because of their different hardware capabilities. Thus, the tablet device has also been used to estimate the constraints of BaseX, which results in the fact that found restrictions are not applying for other devices.
To find the first constraint, which is the maximum size of a BaseX database, different XML files have been created using the XMark tool, explained in Section 4.2.1. Their sizes start by 50 MB and also increases by 50 MB steps. Every step a new database is created and the specific file is added to the database. If the previous step was successful the first five queries of the XMark benchmark suite will be executed on this specific database. The reason for using the first five queries is that those are not so intensive and complex, but they still represent a good example how it could be found in a real application. The selected results can be seen in Table 4.13.

Size	Create	Query 1	Query 2	Query 3	Query 4	Query 5
50 MB	222.7	1.1	3.3	4.4	4.2	1.3
100 MB	484.5	1.6	3.7	8.3	8.2	2.6
150 MB	669.0	2.7	6.0	13.4	13.7	4.2
200 MB	984.0	5.4	10.5	17.9	18.1	5.9
250 MB	1169.5	5.2	9.7	20.3	20.5	6.7
500 MB	2447.5	8.5	19.1	45.3	41.2	13.3
2 GB	8712.8	37.9	81.2	219.5	167.3	22.8

Table 4.13: Measured execution times, in seconds, while creating large databases and subsequently querying them.

The displayed results of Table 4.13 illustrate that creating a large database is combined with a create time effort. Looking at the required times to create each database, shows a linearity which increases about 200 seconds with 50 MB is shown. The query times are illustrating that BaseX is not a choice for applications with many queries combined with a large database in mobile environments, at the moment. However, the results are still proving that it is possible to use BaseX to handle a large database, for mobile devices, in the Android environment. The largest file size, which has been used to find the maximal size of a database, was 2 GB. BaseX needed more than 2.5 hours to create and around 40 seconds to execute Query 1 on the 2 GB database on the Tablet. This is a very long time, but thinking about the dimensions of this file illustrates how this times have been achieved. The 2 GB XML file contains more than 35 million elements, which is not common for the use with mobile

devices. Compared to this amount of elements, the used XML contacts files in Section 4.5 with 20.000 elements[5] has a file size of around 1 MB. Even if there will be no application, in the near future, which uses this amount of data to store with the BaseX Android version it has been shown that it is theoretically possible. In addition to this, it has to be said that the device needs enough free space to create the database in its internal storage indeed.
Additionally to the supported maximum storage size it is not possible to find a minimal CPU and RAM size, because of the huge variety of available Android devices. It can be assumed that if a device has enough hardware resources to run the Android operating system, it also provides enough resources to use the BaseX Android library.
In general it can be said if a device has enough hardware resources it is also possible to use the BaseX Android version with large databases, but it would violate the principles of mobile devices. To avoid this disruption, it is common sense to handle databases with such an amount of data on an external server. Those data could then be accessed and modified with the client version of BaseX, created in Section 3.2.4.

[5] 5000 contact nodes each with 4 elements

Chapter 5

Summary

In this chapter a conclusion about the present thesis and the achieved results and findings is given as well as an overview of possible future work. Additionally a recommendation is proposed on which circumstances BaseX can be used in the Android context instead of the standard Android database SQLite3.

5.1 Conclusion

The goal of migrating the XML database BaseX to the Android platform has been achieved, as an implementation of an Android library as well as a client/server solution. Only a few features, the cryptographic XQuery function for example, could not have been migrated to the Android platform in order to get a working solution. The occurred issues and problems could have been solved, some brought constraints with them which are not restricting BaseX in its fundamental functioning. One of them is the minimal Android API version of the application, that uses the BaseX library, has to be higher than 2.3. As mentioned in the corresponding section there are only less than one percent of the distributed mobile devices using a version lower than 2.3. Another constraint is that also the name of the application has to be passed to the library in order to get it working without any troubles. As a conclusion of the migration it could be said that a positive result has been achieved, even with the above mentioned constraints, because it also could have been impossible to migrate BaseX to the Android platform for unknown circumstances.

The performance of the database operations and the execution of the XMark benchmark queries have been identified by comparing them to results achieved by the BaseX desktop version. The Android version of BaseX has been improved by adjusting the source code of it with techniques which have been proposed to achieve better performance of Android application code. This adjustment also illustrates the big difference between the two platforms, even if it is not clearly visible at the first view. In contrast to the mobile version, applying the same changes to the BaseX desktop version has not improved the performance of it. It has also been shown how difficult it is to compare two different platforms, executed on different devices, to each other. The only similarity is hereby the used programming language and after it is compiled everything differs, beginning from the bytecode format to the execution of it inside a virtual machine. Nevertheless it was possible to optimize the android version and make it up to three times faster for specific conditions. However, the operations are still slow compared to the desktop version of BaseX, but this is mostly a hardware constraint, because of the lack of available hardware resources on a mobile device. Even with this disadvantage the Android version of BaseX is the first available and working solution to execute the XQuery language, with its FLOWR expressions, on Android devices. It is also the first XML database that offers the possibility to be used in the Android operating system.

The comparison of the Android database SQLite3 and BaseX has shown that there is a potential application area for BaseX on the Android platform. Thinking about systems which are using Android as their operating system and are used for collecting a huge amount of data can use the BaseX Android version because of its constant time by inserting data into a database. In contrast to this, it has also been shown that BaseX is no alternative to SQLite3 when a large database is used that will be constantly queried. The measured times that are achieved by querying a SQLite3 database can not be gained by BaseX, it cannot be predicted if, with increasing hardware resources, this circumstance will change. However, depending on how the queries should be processed, BaseX could also being used if the results need to be represented as single string value.
It has been shown that it is even possible to use BaseX on Android with large files, even if this is not recommended because of the big time consumption which is a result of the operations on those large databases.

With the rapid distribution of Android and the devices which are using it as an operating system, it is just a matter of time when the first need of an XML database, or an XQuery processor is needed. At this point the Android version of BaseX can match these requirements with an acceptably manner and can be considered to be used with any Android application.

5.2 Future Work

During the work of the present thesis Google released the new Android version 4.4, codename KitKat. This version of the platform offers different improvements in contrast with the previous versions of Android. Besides the different features that have been implemented to gain a new user experience Google also claims a performance increase with the release of KitKat and an optimization for devices with lower hardware specifications, especially devices with an available RAM of less than 512 MB [18]. In addition to the improvements that Google publishes with the new Android version, there has also been added another feature to the new version, which has not been advertised like the other included features. This feature is also not mentioned in the official Android KitKat website, because it is an early development state and not sophisticated yet. It is the Android Runtime (Art) which should replace the Dalvik virtual machine in the near future [19]. The difference between Art and the DVM is

that ART is not a virtual machine, it is more a compiler that translates the code of an application into native machine language. It can be said that it applies the Just in Time compiler to the whole application and not just for the most executed code parts, as explained in Section 3.1.1. This mechanism is called Ahead of Time compiler, which is done after the application has been installed. This also is the disadvantage of ART, it can take a lot of time to compile the whole application. Nevertheless the speed improvement ART should bring is, compared to the one time consumption the compile step brings, better than executing the application within Dalvik, because there is no additional virtual machine with an interpreter to execute. A lot about the new runtime can not be said, because it is in a very early state and Google has not provided many information about it. Future work could be to investigate the improvements ART brings and if there are adjustments in the BaseX Android library can be done to achieve better runtime values while using ART. The improvements that have been done in Section 4.4 could be kept, because most of them have been done to optimize the JIT mechanism from Dalvik, which is the core principle of ART. But this also needs to be tested and investigated in the future.
On the other side it could be the hardware aspect that can be investigated. Because of the lack of possibilities during the work on this thesis the BaseX Android library could only be tested on one device. A lot of new devices with new hardware specifications will be released every year which each offer different performance abilities. An additional future work could be the focusing on searching for a good hardware specification which is the minimum requirement to achieve assumable execution times of the BaseX library.

Bibliography

[1] S. Amer-Yahia, C. Botev, J. Dorre, and J. Shanmugasundaram. Xquery full-text extensions explained. *IBM Systems Journal*, 45(2):335–351, 2006.

[2] Android Developers. What is android? `http://developer.android.com/guide/basics/what-is-android.html`, 2011. [Online; accessed 27-January-2014].

[3] J. Aycock. A brief history of just-in-time. *ACM Computing Surveys (CSUR)*, 35(2):97–113, 2003.

[4] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, J. Siméon, and M. Stefanescu. Xquery 1.0: An xml query language, 2002.

[5] D. Bornstein. Dalvik vm internals. In *Google I/O Developer Conference*, volume 23, pages 17–30, 2008.

[6] P. Brady. Android anatomy and physiology. In *A talk in Google I/O Conference*, 2008.

[7] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible markup language (xml). *World Wide Web Consortium Recommendation REC-xml-19980210. http://www. w3. org/TR/1998/REC-xml-19980210*, 1998.

[8] E. Burnette. *Hello, Android: introducing Google's mobile development platform.* Pragmatic Bookshelf, 2009.

[9] C. Chen, L. Liu, J. Chen, and C. Zhang. Extracting language strings from xml based on android. *International Conference on Computer Science and Information Technology*, 2011.

[10] B. Cheng and B. Buzbee. A jit compiler for android's dalvik vm. In *Google I/O developer conference*, 2010.

[11] I. D. Craig. *Virtual Machines.* Springer-Verlag, London, 2006.

[12] P. Dubroy. Memory management for android apps. In *Google I/O Development Conference*, 2011.

[13] D. Ehringer. The dalvik virtual machine architecture. *Techn. report (March 2010)*, 2010.

[14] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of android application security. In *USENIX security symposium*, 2011.

[15] G. H. Forman and J. Zahorjan. The challenges of mobile computing. *Computer*, 27(4):38–47, 1994.

[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software.* Pearson Education, 2010.

[17] N. Gandhewar and R. Sheikh. Google android: An emerging software platform for mobile devices. *International Journal on Computer Science and Engineering*, 1(1):12–17, 2010.

[18] Google. Android KitKat. `http://developer.android.com/about/versions/kitkat.html`, 2013. [Online; accessed 23-January-2014].

[19] Google. Introducing Art. `http://source.android.com/devices/tech/dalvik/art.html`, 2013. [Online; accessed 23-January-2014].

[20] C. Grün. *Storing and querying large XML instances.* PhD thesis, University of Konstanz, 2010.

[21] C. Grün, S. Gath, A. Holupirek, and M. H. Scholl. *XQuery full text implementation in BaseX.* Springer, 2009.

[22] J.-M. Kim and J.-S. Kim. Androbench: Benchmarking the storage performance of android-basex mobile devices. In *Frontiers in Computer Education*, pages 667–674. Springer, 2012.

[23] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox. Design of the java hotspot client compiler for java 6. *ACM Transactions on Architecture and Code Optimization (TACO)*, 5(1):7, 2008.

[24] C. Lamb. BerkleyDB Edition on Android. Technical report, Oracle, 2010.

[25] S. Lee and J. W. Jeon. Evaluating performance of android platform using native c for embedded systems. In *Control Automation and Systems (ICCAS), 2012 International Conference on*, pages 1160–1163. IEEE, 2010.

[26] S. Liang and D. Viswanathan. Comprehensive profiling support in the java virtual machine. In *COOTS*, pages 229–242, 1999.

[27] G. Macario, M. Torchiano, and M. Violante. An in-vehicle infotainment software architecture based on google android. In *Industrial Embedded Systems, 2009. SIES'09. IEEE International Symposium on*, pages 257–260. IEEE, 2009.

[28] C. Maia, L. Nogueira, and L. M. Pinho. Evaluating android os for embedded real-time systems. In *Proceedings of the 6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications, Brussels, Belgium*, pages 63–70, 2010.

[29] McObject. McObject Benchmarks Embedded Databases on Android Smartphones. `http://www.mcobject.com/march9/2009`, 2009. [Online; accessed 29-January-2014].

[30] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Evaluating the accuracy of java profilers. In *ACM Sigplan Notices*, volume 45, pages 187–197. ACM, 2010.

[31] G. Nicol, L. Wood, M. Champion, and S. Byrne. Document object model (dom) level 3 core specification. Technical report, W3C, 2001.

[32] M. Nosrati, R. Karimi, and H. A. Hasanvand. Mobile computing: Principles, devices and operating systems. *World Applied Programming*, 2(7), 2012.

[33] H.-S. Oh, B.-J. Kim, H.-K. Choi, and S.-M. Moon. Evaluation of android dalvik virtual machine. In *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, pages 115–124. ACM, 2012.

[34] M. Owens. *The definitive guide to SQLite.* Apress, 2006.

[35] M. Paleczny, C. Vick, and C. Click. The java hotspot tm server compiler. In *Proceedings of the 2001 Symposium on Java TM Virtual Machine Research and Technology Symposium-Volume 1*, pages 1–1. USENIX Association, 2001.

[36] W. Pugh. Compressing java class files. In *ACM SIGPLAN Notices*, volume 34, pages 247–258. ACM, 1999.

[37] J. Roy and A. Ramanujan. Xml schema language: taking xml to the next level. *IT Professional*, 3(2):37–40, 2001.

[38] A. Schmidt, F. Waas, M. Kersten, M. J. Carey, I. Manolescu, and R. Busse. Xmark: A benchmark for xml data management. In *Proceedings of the 28th international conference on Very Large Data Bases*, pages 974–985. VLDB Endowment, 2002.

[39] Y. Shi, K. Casey, M. A. Ertl, and D. Gregg. Virtual machine showdown: Stack versus registers. *ACM Transactions on Architecture and Code Optimization (TACO)*, 4(4):2, 2008.

[40] The Android Open Source Project. VMDebug class. `https://android.googlesource.com/platform/libcore-snapshot/+/ics-mr1/dalvik/src/main/java/dalvik/system/VMDebug.java`, 2007. [Online; accessed 23-January-2014].

[41] A. Tonini, L. Fischer, J. de Mattos, and L. de Brisolara. Analysis and evaluation of the android best practices impact on the efficiency of mobile applications. 2013.

[42] A. R. Tonini, M. Beckmann, J. C. de Mattos, and L. B. de Brisolara. Evaluating android best practices for performance. 2013.

[43] J. Wei. *Android Database Programming.* Packt Publishing Ltd, 2012.

Appendices

Appendix A

The XMark Results

	Q 1	Q 2	Q 3	Q 4	Q 5	Q 6	Q 7	Q 8	Q 9	Q 10
100KB	4.86	5.11	5.2	7.27	2.89	6.51	7.78	16.26	16.87	27.31
200KB	2.12	1.79	2.61	4.36	1.47	6.14	18.23	16.62	23.72	15.4
300KB	0.54	1.19	2.12	1.23	0.74	2.33	6.52	14.37	18.92	5.01
400KB	0.6	1.36	1.57	1.41	0.73	2.96	7.11	23.38	30.04	9.15
500KB	0.46	1.15	1.8	2.12	0.55	3.3	8.56	43.21	58.16	13.3
600KB	0.8	1	2.38	1.66	0.65	4.09	9.27	55.4	67.93	17.4
700KB	0.65	1.21	1.92	1.83	0.59	5.03	10.92	72.23	91.27	21.4
800KB	0.75	1.56	2.9	2.73	0.59	5.25	15.33	92.25	109.28	27.06
900KB	1.08	1.58	3.18	2.92	0.62	5.66	13.72	125.13	161.95	34.43
1000KB	1.14	1.51	3.74	4.76	1.62	7.02	16.06	152.57	183.5	37.03
1100KB	0.82	1.89	2.94	3.7	1.09	7.15	16.05	173.69	222.76	34.14
1200KB	1.39	2.22	3.62	4.17	1.74	8.33	18.31	253.91	286.52	49.62
1300KB	1.17	2.16	4.43	4.04	1.41	8.78	20.97	298.63	355.69	53.89
1400KB	1.18	1.98	4.58	4.09	1.27	9.02	21.22	344.15	400.27	64.99

	Q 11	Q 12	Q 13	Q 14	Q 15	Q 16	Q 17	Q 18	Q 19	Q 20
100KB	11.43	3.96	7.07	11.68	0.75	1.62	2.24	3.61	5.41	6
200KB	12.37	4.18	8.54	10.5	0.7	1.18	1.94	4.89	4.86	2.06
300KB	31.29	10.18	1.47	9.33	0.94	1.26	0.67	1.78	2.01	1.32
400KB	36.7	12.03	1.41	12.37	0.5	0.62	0.91	1.83	3.55	1.29
500KB	76.2	20.65	3.31	10.59	0.45	0.54	1.23	2.73	4.67	1.66
600KB	94.7	25.83	2.66	13.09	1.09	0.7	1.46	1.89	5.37	2.51
700KB	143.31	32.12	2.85	19.5	0.5	0.89	1.68	1.86	5.42	2.73
800KB	182.65	45.38	3.72	21.31	0.83	0.76	1.54	2.24	6.23	2.66
900KB	267.72	56.32	3.42	20.01	0.65	0.76	1.79	3.4	7.11	3.57
1000KB	305.61	57.39	5.72	21.13	0.81	0.95	1.73	4.22	7.59	3.21
1100KB	382.91	76.29	3.82	30.66	1.48	1.26	2.28	3.1	9.38	3.98
1200KB	505.49	149.9	4.61	27.75	1.03	1.17	2.14	3.74	10.06	4.24
1300KB	562.9	127.13	5.22	30.17	1.1	1.2	2.18	3.34	11.11	4.68
1400KB	782.67	206.72	5.11	32.13	1.33	2.05	2.48	3.71	14.25	4.96

Table A.1: The results in milliseconds for the XMark tests using the Laptop.

	Q 1	Q 2	Q 3	Q 4	Q 5	Q 6	Q 7	Q 8	Q 9	Q 10
100KB	11.77	11.43	18.73	25.35	9.53	17.1	38.75	64.62	80.82	99.27
200KB	9.41	15.02	32.9	34.68	11.45	31.55	107.42	326.66	416.43	193.68
300KB	23.22	27.04	66	68.65	23.55	62.6	144.89	656.91	618.17	149.51
400KB	11.51	19.71	41.71	49.59	15.6	54.97	141.07	1007.53	995.21	289.17
500KB	14.96	24.41	53.14	70.74	22.53	119.81	318.78	1721.02	1567.17	646.64
600KB	17.46	25.97	57.24	53.83	20.67	79.4	203.34	2438.53	2424.19	698.62
700KB	37.04	49.44	100.06	114.11	34.26	168.77	406.63	2476.69	3970.64	790.95
800KB	20.3	33.36	79.89	77.4	24.13	94.13	242.13	3707.53	4147.23	799.91
900KB	40.92	60.08	147.54	140.31	40.05	200.28	489.58	4706.09	5248.22	1151.43
1000KB	24.09	39.27	92.41	101.48	30.79	117.88	304.5	4378.18	5388.1	1238.11
1100KB	38.29	69.24	131.48	141.77	44.65	136.53	364.98	6661.28	8060.18	1891.58
1200KB	38	66.3	142.95	176.91	51.4	152.61	407.9	8901.99	10925.29	2659.64
1300KB	40.76	72.72	187.16	229.13	60.39	212.27	679.75	14569.51	15965.77	3935.22
1400KB	64.21	117.78	243.64	276.33	74.03	296.2	793.68	14091.85	14584.44	2882.58

	Q 11	Q 12	Q 13	Q 14	Q 15	Q 16	Q 17	Q 18	Q 19	Q 20
100KB	107.47	41.24	22.16	33.79	9.72	12.12	14.19	10.36	27.67	28.33
200KB	505.47	180.55	62.99	122.67	11.53	23.82	29.93	26.2	71.04	58.33
300KB	650.13	236.88	27.1	98.01	10.94	15.12	20.87	18.87	55.99	41.47
400KB	1032.12	337.08	27.79	143.9	13.89	17.56	26.52	20.87	71.78	51.38
500KB	3239.97	814.04	59.36	140.91	13.12	16.96	35.18	24.16	100.96	61.58
600KB	3744.67	1284.37	72.03	268.95	25.11	31.38	68.66	48.02	186.29	123.09
700KB	4894.18	1121.65	84.06	288.67	15.11	20.07	47.54	38.47	140.98	86.3
800KB	6351.17	1552.26	81.67	278.48	23.92	32.76	77.05	60.02	239.46	138.39
900KB	5992.97	1400.77	64.33	245.36	19.85	26.68	59.46	36.01	163.73	98.83
1000KB	7201.34	1486.59	125.05	280.66	22.93	30.14	62.39	38.59	175.11	106.21
1100KB	10655.3	2489.49	107.36	442.84	34.83	46.57	83.92	57.75	278.28	146.72
1200KB	15368.77	4869.94	142.62	523.53	41.31	51.47	108.75	60.85	282.63	172.56
1300KB	26075.85	6375.99	254.62	866.78	47.85	82.16	173.43	97.94	494.56	291.26
1400KB	20481.03	5526.17	137.2	628.94	53.84	84.06	122.31	64.08	321.4	190.33

Table A.2: The results in milliseconds for the XMark tests using the Tablet.

	Q 1	Q 2	Q 3	Q 4	Q 5	Q 6	Q 7	Q 8	Q 9	Q 10
100KB	2.42	2.24	3.60	3.49	3.30	2.63	4.98	3.97	4.79	3.63
200KB	4.44	8.39	12.61	7.95	7.79	5.14	5.89	19.65	17.56	12.58
300KB	43.00	22.72	31.13	55.81	31.82	26.87	22.22	45.71	32.67	29.84
400KB	19.18	14.49	26.57	35.17	21.37	18.57	19.84	43.09	33.13	31.60
500KB	32.52	21.23	29.52	33.37	40.96	36.31	37.24	39.83	26.95	48.62
600KB	21.83	25.97	24.05	32.43	31.80	19.41	21.94	44.02	35.69	40.15
700KB	56.98	40.86	52.11	62.36	58.07	33.55	37.24	34.29	43.50	36.96
800KB	27.07	21.38	27.55	28.35	40.90	17.93	15.79	40.19	37.95	29.56
900KB	37.89	38.03	46.40	48.05	64.60	35.39	35.68	37.61	32.41	33.44
1000KB	21.13	26.01	24.71	21.32	19.01	16.79	18.96	28.70	29.36	33.44
1100KB	46.70	36.63	44.72	38.32	40.96	19.10	22.74	38.35	36.18	55.41
1200KB	27.34	29.86	39.49	42.42	29.54	18.32	22.28	35.06	38.13	53.60
1300KB	34.84	33.67	42.25	56.72	42.83	24.18	32.42	48.79	44.89	73.02
1400KB	54.42	59.48	53.20	67.56	58.29	32.84	37.40	40.95	36.44	44.35

	Q 11	Q 12	Q 13	Q 14	Q 15	Q 16	Q 17	Q 18	Q 19	Q 20
100KB	9.40	10.41	3.13	2.89	12.96	7.48	6.33	2.87	5.11	4.72
200KB	40.86	43.19	7.38	11.68	16.47	20.19	15.43	5.36	14.62	28.32
300KB	20.78	23.27	18.44	10.50	11.64	12.00	31.15	10.60	27.86	31.42
400KB	28.12	28.02	19.71	11.63	27.78	28.32	29.14	11.40	20.22	39.83
500KB	42.52	39.42	17.93	13.31	29.16	31.41	28.60	8.85	21.62	37.10
600KB	39.54	49.72	27.08	20.55	23.04	44.83	47.03	25.41	34.69	49.04
700KB	34.15	34.92	29.49	14.80	30.22	22.55	28.30	20.68	26.01	31.61
800KB	34.77	34.21	21.95	13.07	28.82	43.11	50.03	26.79	38.44	52.03
900KB	22.39	24.87	18.81	12.26	30.54	35.11	33.22	10.59	23.03	27.68
1000KB	23.56	25.90	21.86	13.28	28.31	31.73	36.06	9.14	23.07	33.09
1100KB	27.83	32.63	28.10	14.44	23.53	36.96	36.81	18.63	29.67	36.86
1200KB	30.40	32.49	30.94	18.87	40.11	43.99	50.82	16.27	28.09	40.70
1300KB	46.32	50.15	48.78	28.73	43.50	68.47	79.56	29.32	44.51	62.24
1400KB	26.17	26.73	26.85	19.57	40.48	41.00	49.32	17.27	22.55	38.37

Table A.3: The factors of the different execution times between the Laptop and the Tablet.

	Q 1	Q 2	Q 3	Q 4	Q 5	Q 6	Q 7	Q 8	Q 9	Q 10
100KB	7.0	8.33	15.35	19.68	7.59	14.25	35.24	54.26	66.68	77.2
200KB	7.56	11.5	26.97	30.91	9.49	29.26	72.9	169	220.73	93.38
300KB	8.85	11.74	32.68	33.87	10.16	38.42	97.04	385.43	500.45	117.98
400KB	9.55	13.9	35.33	41.68	11.07	51.05	126.43	585.35	768.09	257.35
500KB	11.7	17.88	43.24	42.93	14.37	64.97	166.13	1222.6	1572.54	499.29
600KB	13.85	19.07	47.51	44.61	15.25	70.09	182.86	1734.95	2081.51	668.64
700KB	15.16	21.05	55.69	54.26	16.75	85.75	216.67	2035.3	2508.23	642.33
800KB	15.65	24.3	59.44	59.96	17.18	88	228.75	2188.01	2728.3	657.47
900KB	17.91	27.74	62.8	65.91	19.6	106.65	267.39	3399.04	4126.52	970.42
1000KB	19.01	29.72	66.16	76.75	23.74	111.87	295.01	4021.91	4883.11	1235.96
1100KB	20.35	31.79	76.56	84	25.48	117.43	315.88	4585.87	5829.68	1129.57
1200KB	22.85	35.21	85.63	99.39	28.83	131.1	355.77	6528.22	7789.61	1728.3
1300KB	24.5	37.8	82.73	113.18	29.44	143.43	384.97	7750.3	9327.97	1807.03
1400KB	25.9	41.1	93.35	106.96	29.87	154.84	412.87	8384.97	10487.64	2040.58

	Q 11	Q 12	Q 13	Q 14	Q 15	Q 16	Q 17	Q 18	Q 19	Q 20
100KB	84.9	31.7	19.28	30.02	6.98	9.24	10.49	8.48	24.29	23.74
200KB	253.99	78.09	33.17	64.94	8.02	11.41	14.55	13.3	36.65	28.13
300KB	659.1	189.05	21.6	90.6	9	13.36	16.02	16.17	46.75	33.75
400KB	993.71	290.28	23.22	136.07	10.8	13.8	20.38	18.24	59.08	40.1
500KB	1956.81	534.3	48.7	135.28	10.31	13.39	28.04	23	85.08	49.59
600KB	2687.83	733.3	40.46	153.16	11.04	14.86	33.38	21.97	96.92	59.53
700KB	3116.3	835.63	42.17	203.43	11.62	15.38	35.06	21.91	104.24	60.3
800KB	3623.65	1002.53	47.79	205.59	14.92	19.1	33.66	25.08	110.44	63.2
900KB	5394.22	1186.3	46.14	228.14	15.01	19.71	44.79	30.21	132.01	74.61
1000KB	6076.93	1291.87	86.85	257.84	17.48	23.16	47.08	31.89	144.5	78.27
1100KB	7166.13	1630.67	52.3	257.17	21.05	26.91	45.89	34.36	152.13	82.18
1200KB	10153.62	3276.55	70.67	320.52	22.03	29.2	57.98	38.2	174.83	99.57
1300KB	12085.54	2915	84.86	354.09	22.91	30.21	60.91	39.67	192.96	104.55
1400KB	14108.39	3668.67	72.23	361.54	22.73	30.39	63.53	42.81	207.18	111.93

Table A.4: The results in milliseconds for the XMark benchmark tests using the optimized BaseX Android version.

	Q 1	Q 2	Q 3	Q 4	Q 5	Q 6	Q 7	Q 8	Q 9	Q 10
100KB	68%	37%	22%	28%	25%	20%	9%	19%	21%	28%
200KB	24%	30%	21%	12%	20%	7%	47%	93%	88%	107%
300KB	162%	130%	101%	102%	131%	62%	49%	70%	23%	26%
400KB	20%	41%	18%	18%	40%	7%	11%	72%	29%	12%
500KB	27%	36%	22%	64%	56%	84%	91%	40%	0%	29%
600KB	26%	36%	20%	20%	35%	13%	11%	40%	16%	4%
700KB	144%	134%	79%	110%	104%	96%	87%	21%	58%	23%
800KB	29%	37%	34%	29%	40%	6%	5%	69%	52%	21%
900KB	128%	116%	134%	112%	104%	87%	83%	38%	27%	18%
1000KB	26%	32%	39%	32%	29%	5%	3%	8%	10%	0%
1100KB	88%	117%	71%	68%	75%	16%	15%	45%	38%	67%
1200KB	66%	88%	66%	78%	78%	16%	14%	36%	40%	53%
1300KB	66%	92%	126%	102%	105%	48%	76%	87%	71%	117%
1400KB	147%	186%	161%	158%	147%	91%	92%	68%	39%	41%

	Q 11	Q 12	Q 13	Q 14	Q 15	Q 16	Q 17	Q 18	Q 19	Q 20
100KB	26%	30%	14%	12%	39%	31%	35%	22%	13%	19%
200KB	99%	131%	89%	88%	43%	108%	105%	96%	93%	107%
300KB	-1%	25%	25%	8%	21%	13%	30%	16%	19%	22%
400KB	3%	16%	19%	5%	28%	27%	30%	14%	21%	28%
500KB	65%	52%	21%	4%	27%	26%	25%	5%	18%	24%
600KB	39%	75%	78%	75%	127%	111%	105%	118%	92%	106%
700KB	57%	34%	99%	41%	30%	30%	35%	75%	35%	43%
800KB	75%	54%	70%	35%	60%	71%	128%	139%	116%	118%
900KB	11%	18%	39%	7%	32%	35%	32%	19%	24%	32%
1000KB	18%	15%	43%	8%	31%	30%	32%	21%	21%	35%
1100KB	48%	52%	105%	72%	65%	73%	82%	68%	82%	78%
1200KB	51%	48%	101%	63%	87%	76%	87%	59%	61%	73%
1300KB	115%	118%	200%	144%	108%	171%	184%	146%	156%	178%
1400KB	45%	50%	89%	73%	136%	176%	92%	49%	55%	70%

Table A.5: The performance improvements in percents.

Appendix B

The Query and Modify Statements

```sql
SELECT * FROM contacts ORDERED BY name;
```

Listing B.1: Statement 1 as SQL statement

```
for $x in .//contact/name order by $x return $x/..
```

Listing B.2: Statement 1 as XQuery code

```sql
SELECT * FROM contacts WHERE email LIKE '%com';
```

Listing B.3: Statement 2 as SQL statement

```
.//contact[ends-with(email, 'com')]
```

Listing B.4: Statement 2 as XQuery code

```sql
SELECT * FROM contacts WHERE name LIKE '%Meier%';
```

Listing B.5: Statement 3 as SQL statement

```
.//contact/name[text() contains text {'Meier'}] /..
```

Listing B.6: Statement 3 as XQuery code

```sql
DELETE FROM contacts WHERE email LIKE '%org';
```

Listing B.7: Statement 4 as SQL statement

```
delete node .//contact[ends-with(email, 'org')]
```

Listing B.8: Statement 4 as XQuery code

```sql
INSERT INTO contacts (name, city, address, email) values ("Mueller"
    , "Konstanz", "Brauneggerstr␣62", "mueller@htwg.de");
```

Listing B.9: Statement 5 as SQL statement

```
let $insert :=
<contact>
  <name>Mueller</name>
  <city>Konstanz</city>
  <address>Brauneggerstr 62</address>
  <email>mueller@htwg.de</email>
</contact>
return
insert node $insert into /DocumentElement
```

Listing B.10: Statement 5 as XQuery code

```
update contacts set name = "Meier" where name = "Mueller";
```

Listing B.11: Statement 6 as SQL statement

```
replace value of node .//contact[name = 'Mueller']/name with 'Meier
    '
```

Listing B.12: Statement 6 as XQuery code