NanoBase: A tiny relation database manager for the JME CLDC/MIDP Platform

Leonardo Eloy
Universidade de Fortaleza,
leonardo.eloy@gmail.com

Vitor Vasconcelos
Universidade de Fortaleza,
vitor.vasconcelos@gmail.com

José Maria Monteiro
Universidade de Fortaleza,
Av. Washington Soares, 1321
- Mestrado em Informática
Aplicada
Edson Queiroz
60811341 - Fortaleza, CE -
Brasil, monteiro@unifor.br

Ângelo Brayner
Universidade de Fortaleza,
Av. Washington Soares, 1321
- Mestrado em Informática
Aplicada
Edson Queiroz
60811341 - Fortaleza, CE -
Brasil, brayner@unifor.br

Resumo
Os recentes avanços obtidos na tecnologia de computação móvel possibilitam o desenvolvimento de novas e sofisticadas aplicações, as quais podem agora armazenar dados no próprio dispositivo móvel. A plataforma JME vem se consolidando como um padrão para o desenvolvimento de aplicações em dispositivos móveis, também denominadas de aplicações embarcadas. Nesta plataforma, a API RMS é responsável pela persistência dos dados. Esta API possibilita a recuperação, inserção, alteração e exclusão de registros em arquivos representados por vetores (arrays) de bytes. Tal característica torna a programação bastante complexa e de baixa produtividade, principalmente no tocante à persistência de dados. Além disso, como nenhuma estrutura de índice é disponibilizada, o acesso seletivo aos registros é bastante demorado, podendo até mesmo inviabilizar o desenvolvimento de determinadas aplicações. Este trabalho apresenta o NanoBase, um mecanismo que fornece funcionalidades de um SGBD relacional para a plataforma JME. O mecanismo proposto fornece aos programadores uma visão relacional dos dados. Outras propriedade importantes do NanoBase são o suporte para execução de consultas, expressas através da linguagem SQL, e a criação de diversas estruturas de índices, como árvores B+ e índices hash. A fim de comprovar os benefícios da utilização do NanoBase, foram realizados diversos testes de desempenho e uma análise comparativa com outras soluções.

Palavras-chave: *Bancos de dados móveis. Processamento de consulta. Plataforma JME CLDC/MIDP.*

Abstract
Increasing advances in mobile computing made possible the development of new and sophisticated applications, which can store data in the mobile devices. The JME platform has emerged as a standard for developing applications for mobile devices, the so-called embedded applications. The RMS API is responsible for ensuring data persistence in the JME platform. In other words, this API provides support for manipulating records stored in files. However, the RMS API "views" files as a set of arrays of bytes. Such a feature makes the process of accessing data in JME platform a very complex and inefficient. Moreover, since no index structure is supported in JMS platform, the access to subset of records in a file tends to be quite inefficient. This work presents NanoBase, a mechanism which provides relational database features for the JME/CLDC platform. A key feature of Nanobase is the ability of processing SQL queries. Moreover, NanoBase can build and manipulate different index structures, such as B+ trees and hash indexes, for efficiently accessing data in JME platform. In order to show the benefits of using NanoBase, we have run extensive performance tests and made comparative analysis with other solutions.

Keywords: *Mobile database. Query processing. JME CLDC/MIDP platform*

1 Introduction

Recent advances in mobile device technology (such as PDAs and smartphones) are contributing to reduce the cost of such equipments, making them even more accessible to a larger group of people. It is estimated, for example, that there are more than one billion people who are users of wireless communication services in the world, and three billion people will use mobile devices in 2010 (Hoschka, 2007). These advances have contributed to increase the computing and storage capacity of mobile equipments. In the beginning of 2006, Seagate, a storage devices manufacturer, presented a 12 GB hard drive for mobile devices (Seagate ST1.3 Series) (Rojas, 2007). These Mini HDs use flash memories (e.g., Smart

Media and MultiMediaCard) to storage data in a permanent manner. This kind of non-volatile memory uses the EEPROM (Electrically Erasable Programmable Read-Only Memory) technology, and it is characterized by having small physical size, reduced weight and shock resistance. However, read and write operations over EEPROMS are relatively slow. Such memory devices have made possible the development of new and sophisticated applications, which require larger storage area, such as home banking, billing, ticket reservation, patient's data monitoring (Monteiro, 2005). Such applications need to manipulate a growing data volume, stored locally in mobile devices.

The JME (Java Micro Edition) platform has consolidated as a standard regarding the development of applications for mobile devices. In order to allow data management, the JME platform provides the RMS (Record Management System) API. This API provides an interface between users/applications and the file system provided by the JME/CLDC platform. A file, called "RecordStore", is a set of byte arrays, where a byte array represents a record of the file. To each record in a RecordStore is associated an identifier. Records can only be accessed sequentially by means of the record's id. Therefore, to access a specific record R with id K, it is necessary to read all records which precede R, i.e., records whose ids are less than K. Moreover, the RMS API does not support the use of any index structure to access data in RecordStore. Such characteristics make data management in JME/CLDC platform a very complex and inefficient task, which should be carried out by applications, since there is no query language in that platform. On the other hand, mobile devices can already store relatively large data volume. For that reason, it is mandatory to provide a more efficient data management system to the JME/CLDC.

With the aim of facilitating the development of data-centric applications for mobile devices, we envision that it is essential to build mechanisms for providing DBMS features to the JME/CLDC platform. Such mechanisms should allow, for instance, the use of a query language, such as SQL, and the use of indexes, to speed up the data access process.

This paper presents NanoBase, a mechanism which provides a relational view over data stored through the JME/CLDC platform, a query processor for processing SQL's DDL/DML expressions. Thus, by using NanoBase it is possible to define and maintain integrity constraints, such as key constraint (primary key) or referential integrity (foreign key). Another key feature of NanoBase is the support for defining (building) different index structures (B+ Tree, Dynamic Hashing, Bitmap Index and Kd Trees) to access data in a more efficient way. Thus, the proposed mechanism viewed as a tiny relational database manager to run over the JME/CLDC platform.

This paper is organized in the following way: section 2 presents Nanobase. Section 3 discusses related work. On section 4 we present the results of the tests carried through. Section 5 concludes this work and points directions to future works.

2 NanoBase

As already mentioned, NanoBase is in fact a relational database manager. It behaviors like an interface between application/users and the file system provided by the JME/CLDC platform. For that reason, we can not classify NanoBase as a system, since it does not have any active component such as the scheduler or buffer manager in a database system.

Therefore, the strategy used to implement NanoBase was to build multiple layers, each of which being composed by different APIs. Figure 1 depicts NanoBase's architecture, which comprises the following APIs: Access Manager, Query Engine and Storage Manager. Besides those APIs, there is another component belonging to the NanoBases's architecture, Metadata structure. Next, we describe those components.

The Access Manager API consists of the layer between applications and NanoBase. In other words, applications invoke directly this API, which in turn invokes classes and methods of the others APIs (Query Engine and Storage Manager). Using a JDBC-like syntax, the Access Manager API provides a set of classes and methods that makes possible for the developer to submit SQL DDL and DML clauses and to manipulate the result of submitted queries. Since JDBC is actually a standard for data access, the Access Manager API learning curve is low and its use is quite easy.

The Query Engine API is composed by a query processor, optimizer and executor. Its main function is to parse SQL clauses in optimized execution plans and proceed to execute such plans. The Storage Manager API is responsible for providing a relation view over stored data, executing the data storage and retrieval using RMS or FileChannel (JSR 75) in a transparent manner.

The Metadata structure represents the metadata used in the RMS or FileChannel mapping to the relational model and vice-versa.

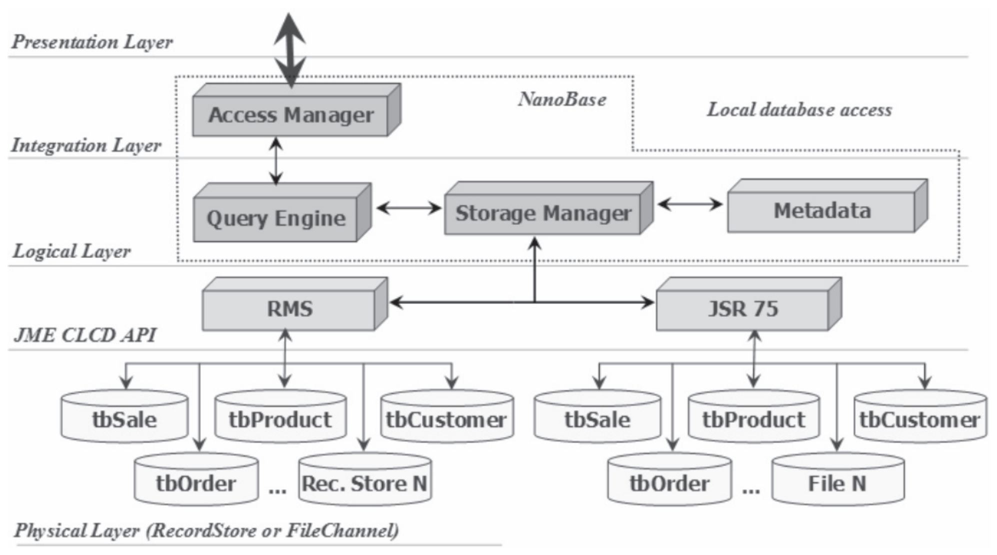


Figure 1: NanoBase's architecture.

2.1 Query engine design

The query engine connects all the NanoBase internal APIs. It's responsible by the lexical, syntactical and semantic analysis. Also, it makes the SQL translation (for an internal representation) and optimization. Figure 2 shows the steps defined and implemented in the query engine design.

The used SQL grammar consists in a PL-SQL simplification. This grammar is concise (to minimize the overhead in the parsing time) and expressive (to make possible the use of DDL/DML clauses and hints expressions.

In order to design and implement the NanoBase Query Engine we use JavaCC, one of the most popular parser generator. A parser is a program that receives a text and says if this text is correct in accordance with a previous defined grammar. The JavaCC allows add java code together the specified grammar to make semantic analysis. Then, the product generated by JavaCC can do lexical, syntactical and semantic analysis for SQL clauses written in our simplified SQL grammar. However, this parser doesn't run in JME platform (only in JSE platform). For this reason we did some corrections in the java code generated by JavaCC. Thus, the modified parser can now runs in JME platform. This parser receives an SQL clause, makes lexical, syntactical and semantic analysis, and, finally, builds a structure called "QueryElements" (a java object that represents the entire SQL clause).

Now, it was necessary translates the SQL Clause (represented by a "QueryElements" object) in a relational algebraic expression. This is important because a relational algebraic expression defines a sequence of steps that retrieve the required data (by the SQL clause). Then, we defined the "Algebraic Tree Builder". This component is a java class that receives a "QueryElements" object (representing an analyzed SQL clause) and returns an "Algebraic Tree" equivalent to the received object.

The "Optimizer" receives an "Algebraic Tree" (object) and try find another equivalent "Algebraic Tree" that runs more fast. The NanoBase optimizer uses the heuristics based optimization. The raison for this choice is that a cost based optimization is unfeasible for devices with low processing resources. In this context, 18 heuristics where carefully selected, implemented and ranged by their relevance in the optimization process. This component returns the optimized algebraic tree.

Finally, we implemented the "Executor". This java class has algorithms to execute each relational operation (projection, selection, join, etc). This component receives the optimized algebraic tree, cover this tree recursively, running for each relational operation the corresponding algorithm. After this, the "Executor" returns a "ResultSet" object containing the required data.

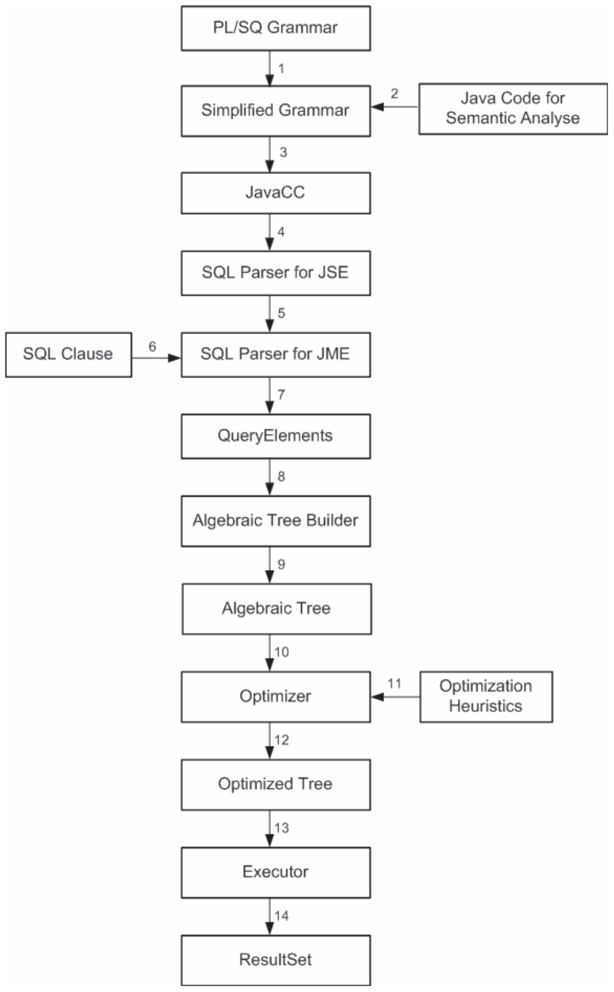


Figure 2: Query Engine Design.

2.2 Access manager API

The Access Manager API consists in a set of classes and interfaces that sends user SQL clauses to Storage Manager. This API is formed by 4 main classes, in a JDBC-Like manner: *DatabaseConfiguration*, *Connection*, *Statement* and *ResultSet*.

Connection: Represents a connection with the Storage Manager.

Statement: Used to sends a SQL clause to the Storage Manager. The *executeQuery* method of *Statement* class uses the Query Engine API to process and to executes a SQL (DML) clause.

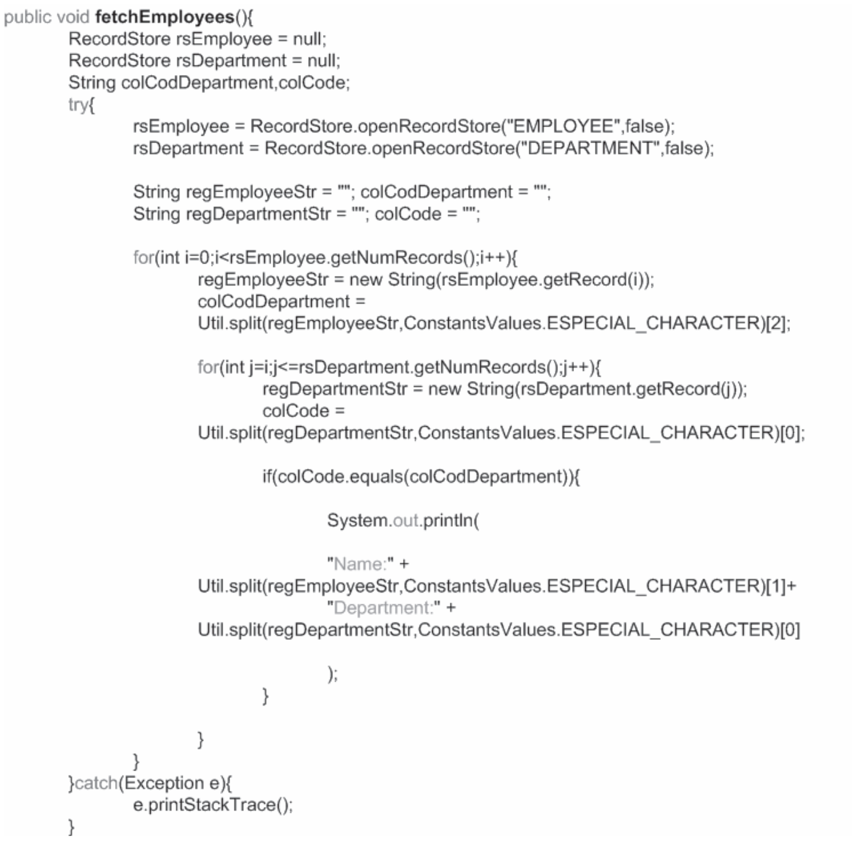
ResultSet: The data set equivalent to a submitted SQL (DML) clause (submitted by *executeQuery* method).

2.3 Storage manage API

Portable devices technology is in constant evolution. The storage capacity and the storage APIs changes quickly. Some years ago the unique API to make possible data storage in JME/CLDC platform was RMS API. Nowadays, FileConnection API (JSR 75) is another API to data storage in JME/CLDC platform. Probably, in the future, news APIs to data storage in mobile devices will appear. Thinking this, the NanoBase architecture allows that different APIs (to store data) be easily jointed. For this, the developer need only implements 3 interfaces (Storage, Table and TablePool). The main interface is Table, which is responsible to provide a data relational view (abstraction).

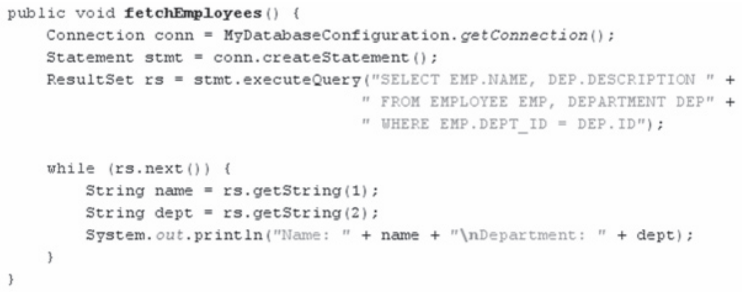
2.4 Running example

To illustrate the use of NanoBase and its benefits, we will use the following scenario: consider wo RecordStores which store Employee and Department data. The columns in the Employee RecordStore are: code, name and department_code. The columns defined for the Department RecordStore are: code and description. Suppose that an application need to execute the following task, named Task 1: fetch the names of all Employees and the description of the departments they work. Next, we will show how this task would be implemented using RMS and how it was implemented using NanoBase. Figure 3 illustrates how Task 1 would be implemented using the RMS API. In this case, we would have to cover, for every record on the Employee RecordStore, all records of the Department RecordStore, verifying if the column department_code of the Employee RecordStore is equal to the "column" code of the Department RecordStore. Figure 4 shows the code necessary to execute Task 1 using NanoBase. Notice that for such the Connection, Statement and ResultSet classes were used.



```
public void fetchEmployees(){
        RecordStore rsEmployee = null;
        RecordStore rsDepartment = null;
        String colCodDepartment,colCode;
        try{
                rsEmployee = RecordStore.openRecordStore("EMPLOYEE",false);
                rsDepartment = RecordStore.openRecordStore("DEPARTMENT",false);

                String regEmployeeStr = ""; colCodDepartment = "";
                String regDepartmentStr = ""; colCode = "";

                for(int i=0;i<rsEmployee.getNumRecords();i++){
                        regEmployeeStr = new String(rsEmployee.getRecord(i));
                        colCodDepartment =
                        Util.split(regEmployeeStr,ConstantsValues.ESPECIAL_CHARACTER)[2];

                        for(int j=i;j<=rsDepartment.getNumRecords();j++){
                                regDepartmentStr = new String(rsDepartment.getRecord(j));
                                colCode =
                        Util.split(regDepartmentStr,ConstantsValues.ESPECIAL_CHARACTER)[0];

                                if(colCode.equals(colCodDepartment)){

                                        System.out.println(

                                        "Name:" +
                        Util.split(regEmployeeStr,ConstantsValues.ESPECIAL_CHARACTER)[1]+
                                        "Department:" +
                        Util.split(regDepartmentStr,ConstantsValues.ESPECIAL_CHARACTER)[0]

                                        );
                                }

                        }
                }
        }catch(Exception e){
                e.printStackTrace();
        }
```

Figure 3: Task 1 using RMS API.



```
public void fetchEmployees() {
    Connection conn = MyDatabaseConfiguration.getConnection();
    Statement stmt = conn.createStatement();
    ResultSet rs = stmt.executeQuery("SELECT EMP.NAME, DEP.DESCRIPTION " +
                                " FROM EMPLOYEE EMP, DEPARTMENT DEP" +
                                " WHERE EMP.DEPT_ID = DEP.ID");

    while (rs.next()) {
        String name = rs.getString(1);
        String dept = rs.getString(2);
        System.out.println("Name: " + name + "\nDepartment: " + dept);
    }
}
```

Figure 4: Task 1 using NanoBase

3 Related work

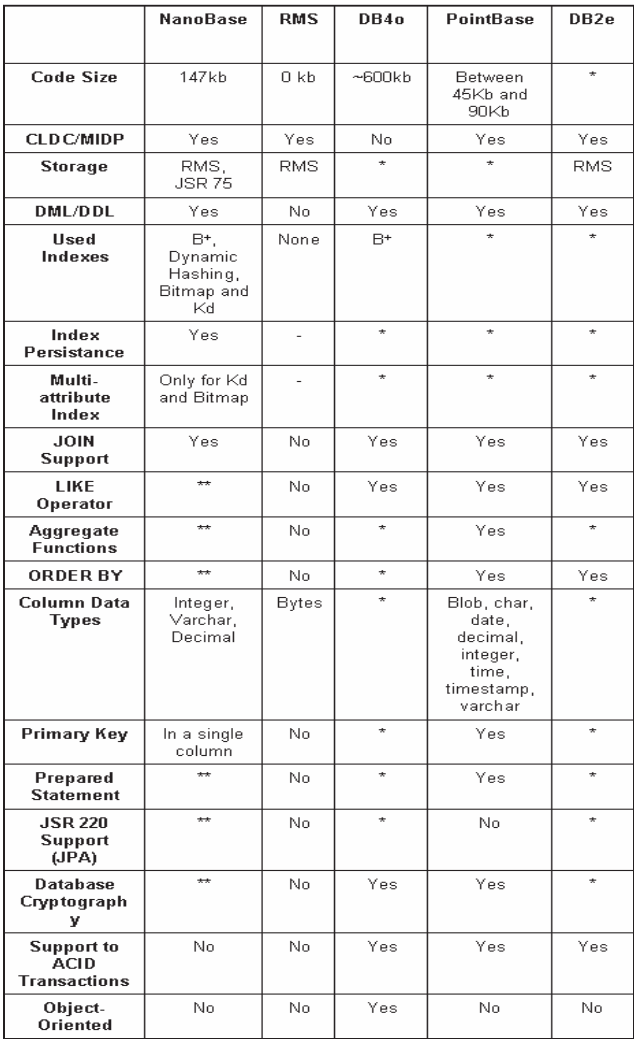
This section will present a brief description of some proprietary tools related to data persistence in JME CLDC/MIDP platform.

PointBase Micro Edition: Provides a compact architecture, consuming only 90Kb in the JME CDC and 45Kb for JME CLDC/MIDP platform. This relational database is simple, having as strong points relevant features for mobile devices, such as column and database cryptography, bi-directional synchronization with any device. Furthermore provides an API which allows the developer to make JDBC connections and also provides a MIDlet that accesses a database. The classes made available have a similar terminology as the JDBC API. Last Version: 5.5. Manufacturer: DataMirror.

IBM DB2e: Provides an interface named FastRecordStore, which uses the RMS API, and can synchronize with a database server. Make possible the execution in many platforms, such as Windows Mobile, Palm OS, Linux and JME CLDC/MIDP. Seems to be efficient, but presents a very difficult to use API. Last version: 9.1. Manufacturer: IBM.

DB4o: Object-oriented database. Executes some JME dialects that supports reflection, such as CDC, PersonalProfile, Symbian, Palm OS, Savage and Zaurus. There is a already a on-going study to make this application available in JME dialects that do not support reflection as CLDC/MIDP. Has strong replication and cryptography features, and implements many query languages, such as SQL, Query by Example (QbE) and Simple Object Data Access (SODA). Last Version: 6.2. Manufacturer: db4objects.

Figure 5 presents the results of a comparative analysis amongst the main tools used for data storage and retrieval in mobile devices.



	NanoBase	RMS	DB4o	PointBase	DB2e
Code Size	147kb	0 kb	~600kb	Between 45Kb and 90Kb	*
CLDC/MIDP	Yes	Yes	No	Yes	Yes
Storage	RMS, JSR 75	RMS	*	*	RMS
DML/DDL	Yes	No	Yes	Yes	Yes
Used Indexes	B+, Dynamic Hashing, Bitmap and Kd	None	B+	*	*
Index Persistance	Yes	-	*	*	*
Multi-attribute Index	Only for Kd and Bitmap	-	*	*	*
JOIN Support	Yes	No	Yes	Yes	Yes
LIKE Operator	**	No	Yes	Yes	Yes
Aggregate Functions	**	No	*	Yes	*
ORDER BY	**	No	*	Yes	Yes
Column Data Types	Integer, Varchar, Decimal	Bytes	*	Blob, char, date, decimal, integer, time, timestamp, varchar	*
Primary Key	In a single column	No	*	Yes	*
Prepared Statement	**	No	*	Yes	*
JSR 220 Support (JPA)	**	No	*	No	*
Database Cryptograph y	**	No	Yes	Yes	*
Support to ACID Transactions	No	No	Yes	Yes	Yes
Object-Oriented	No	No	Yes	No	No

Figure 5: Comparative Analysis Between DBMSs for Móbile Devices.

3.1 Customizing and extensibility

Most of the studied products provide a fixed functionality set. However, some offered functions are not necessary for a specific application. Most applications do not use the whole features, but a reduced, varying function set. Then, mobile applications can be supported by a general purpose database management system which is heavyweight, feature-laden and costly in installation and maintenance, or, otherwise, by a lightweight, customized system.

Obviously, a general "all in one" DBMS cannot fulfill the requirements of customizing and extensibility, which are very important for small devices with limited resources. A viable approach for solving this problem could be a customizable DBMS allowing to choose and combine features from the set of available features.

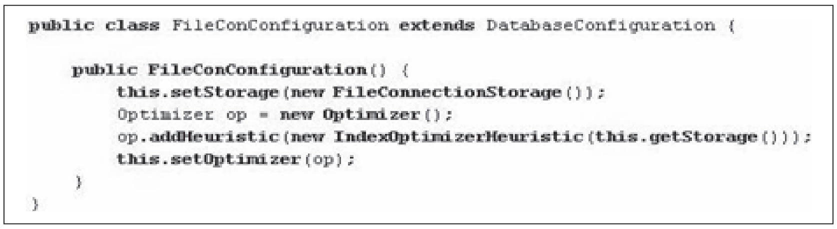
In general, customizable approaches to create database management systems utilize a parameterizable DBMS which can be adjusted by parameters or modified on code level to change its behavior. For this reason very detailed knowledge concerning the specific DBMS and DBMS technology in general is mandatory.

Another idea consists of a set of modules which can be combined depending on the requirements of the application. In addition, new modules can be easily plugged in without affecting the application.

The NanoBase allows some customizing options, in order to adapt its use to the applications requirements and the hardware resources available in the portable devices: data persistence can be made using the RMS API or FileChannel (JSR 75), the developer can choose any sub-set of the 18 optimization heuristics implemented in NanoBase (allowing the tuning between the optimization process quality and the overhead of this process), also, different indexes structures (Tree, Dynamic Hashing, Bitmap Indexes and Kd Trees) can be created and used by hints, besides, an index can be persistent or entire stored in volatile memory.

3.2 The databaseconfiguration class

Using the DatabaseConfiguration class a developer can set some customizing options. Throughout of this class a programmer can obtain a Connection with the set characteristics. The figure 6 shows a java code (using Access Manager API) that defines a configuration class (denoted FileConConfiguration) that extends the basic Database Configuration class and set FileConnection with the data storage API (line 3). Then, instantiates a Optimizer object (line 4). Next, the code instantiates one of the eighteen implemented heuristics (IndexOptimizerHeuristic) and add this heuristics in the Optimizer object (line 5). In line 6, the code add the created optimizer in the current configuration (FileConConfiguration class).



```
public class FileConConfiguration extends DatabaseConfiguration {

    public FileConConfiguration() {
        this.setStorage(new FileConnectionStorage());
        Optimizer op = new Optimizer();
        op.addHeuristic(new IndexOptimizerHeuristic(this.getStorage()));
        this.setOptimizer(op);
    }
}
```

Figure 6: Customizing the Storage API and the Optimizer Heuristics.

Now, observes the following code, which creates a FileConConfiguration object and after instantiates a Connection object. Notes that this connection uses the configurations set in FileConConfiguration class (the storage API is FileChannel, defined by FileConnectionStorage class, and IndexOptimizerHeuristic will be the unique optimizing heuristic used by the Query Engine).

FileConConfiguration conf = new FileConConfiguration();

Connection conn = conf.getConnection();

4 Experimental results

With the purpose to prove the NanoBase benefits and evaluate the performance gains obtained by the use of implemented index, we carried through a performance comparison between NanoBase and a sequential search via RMS. For these tests we used a RecordStore which records having only two attributes: an identifier and a numeric value. The tests were repeated for different amounts of records: 100, 200, 400, and 800 records. The metric used in the performance evaluating was the response time in milliseconds (ms). It was used two queries: one using exact match and another using range values, the latter recovering 20 and 80% of the records, respectively. The device used was a Nokia 6111 (Heap size: 2 MB and Shared memory for Storage: 22 MB). The tests results are illustrate in figures 7 and 8.

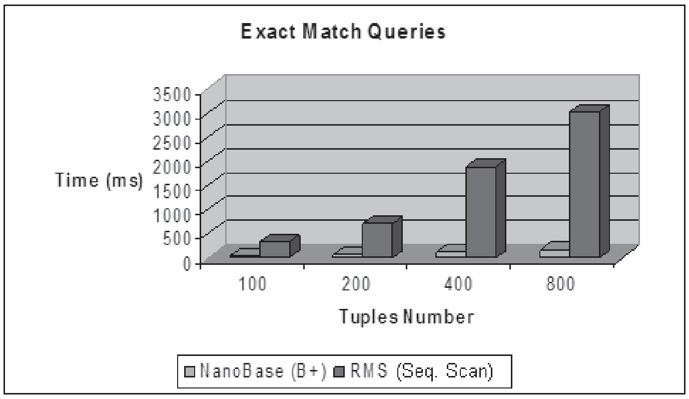


Figure 7: Performance Comparison between NanoBase and RMS APO: Exact Match Queries.

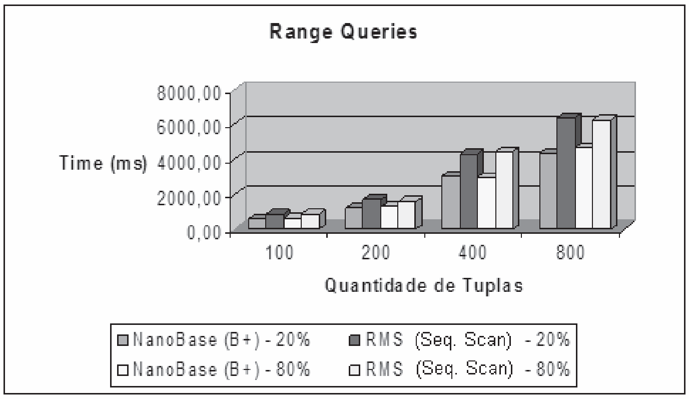


Figure 8: Performance Comparison between NanoBase and RMS APO: Range Queries.

5 Conclusion and future work

In this paper we present NanoBase, a query processor for the JME CLDC/MIDP platform, that making possible the use of DDL/DML clauses, integrity constraints, besides having many index structures. The existent indexes can be used directly throughout NanoBase API. The proposed tool makes the data access much more efficient, which makes possible the development of applications that were impracticable due to low productivity and delay on data retrieval. As future works, we pretend to add support to cryptography, JSR 220 (JPA) and data synchronization between mobile devices and database servers.

References

AHAMED, S.; VALLECHA, S. Component-based embedded database for mobile embedded systems. In: INTERNATIONAL CONFERENCE OF INFORMATION TECHNOLOGY: Coding and Computing, 4., 2004, Las Vegas. *Proceedings*... Las Vegas: ITCC, 2004. v.1, p. 534-538.

BOLCHINI, C.; SCHREIBER, F.A.; TANCA, L. A methodology for a very small data base design. *Information Systems*, Oxford, v. 32, n. 1, p. 61-82. 2007.

BUCHMANN, E.; HÖPFNER, H.; SATTLER, K. U. An extensible storage manager for mobile dbms. In: BALTIC CONFERENCE, 2002, Tallinn. *Proceedings*... Tallinn: BalticDB&IS, 2002. p. 103-116.

HOSCHKA, P. *MWI*: *Leading mobile web access to its full potential*. Disponível em: <http://www.w3.org/blog/MWITeam/2007/06/14/title_21/>. Acesso em 4 jun. 2007.

KARLSSON, J. S. et al. IBM DB2 Everyplace: a small footprint relational database system. In: IEEE INTERNATIONAL CONFERENCE ON DATA ENGINEERING, 2001, Heidelberg. *Proceedings*... Heidelberg: IEEE, 2001. p. 230 -232.

KURAMITSU, K.; SAKAMURA, K. Towards ubiquitous database in mobile commerce. In: ACM INTERNATIONAL WORKSHOP ON DATA ENGINEERING FOR WIRELESS AND MOBILE ACCESS, 2., 2001, Santa Barbara. *Proceedings*...Santa Barbara, CA: ACM, 2002. p. 84–89.

MONTEIRO, J. M.; BRAYNER, A. Controlling concurrency in mobile computing environments with broadcast-based data dissemination. In: EUROPAR CONFERENCE, 2005, Lisboa. *Proceedings*...Lisboa: EUROPAR , 2005. p. 1069-1079.

MONTEIRO, J. M. et al. A mechanism for replicated data consistency in mobile computing environment. IN: ACM SYMPOSIUM ON APPLIED COMPUTING, 22., 2007, Seoul. *Proceedings*... Seoul: SAC-ACM, 2007. p. 914-919.

ROJAS, P. *Seagate announces first 12GB 1-inch hard drive*. Disponível em: <http://www.engadget.com/2006/02/13/seagate-announces-first-12gb-1-inch-hard-drive/>

Acesso em: 4 jun. 2007.

SIQUEIRA, F.; BRAYNER, A. Demo: mobile database administrator-mdba. In: INTERNATIONAL CONFERENCE ON MOBILE DATA MANAGEMENT, 6., 2005, Ayia Napa. *Proceedings*... Ayia Napa: Cyprus, 2005. p. 316-318.

SIQUEIRA, F.; BRAYNER, A. Remote database administration in mobile computational environments. of the 1st INTERNATIONAL WORKSHOP ON MOBILITY AWARE TECHNOLOGIES AND APPLICATIONS, 1., 2004, Florianópolis. *Proceedings*... Florianopolis: MATA , 2004. p. 137-146.

SOBRE OS AUTORES

Leonardo Eloy

Graduando em Ciência da Computação pela Universidade de Fortaleza. Atualmente trabalha com o desenvolvimento de aplicações de Business Intelligence, ERP e com pesquisas na área de Banco de Dados.

Vitor Vasconcelos

Bacharel em Informática pela Universidade de Fortaleza em 2006. Atualmente trabalha como Analista de Sistemas no desenvolvimento de aplicações Web e com pesquisas na área de Banco de Dados.

José Maria Monteiro

Graduado em Ciência da Computação pela Universidade Federal do Ceará em 1998, M.Sc. Ciência da Computação pela Universidade Federal do Ceará em 2001. Doutorando em Informática pela PUC-Rio.. Atualmente ocupa o posto de professor assistente junto ao Centro de Tecnologia da Universidade de Fortaleza onde atua em nível de graduação e pós graduação.

Ângelo Brayner

Angelo Brayner concluiu o doutorado em Ciência da Computação pela Universität Kaiserslautern, Alemanha, em 1999. Atualmente é professor titular da Universidade de Fortaleza, onde é responsável pela cadeira de Banco de Dados dos cursos de graduação e Mestrado em Ciência da Computação da UNIFOR. É autor do livro Transaction Management in Multidatabases Systems, publicado pela Shaker-Verlag, Alemanha, em 1999. Prof. Angelo Brayner é autor de vários artigos publicados em periódicos e conferências internacionais e nacionais.