Logical and physical data structures for Flash memories for lightweight, portable databases

C. Bolchini, A. Giusti, F. Furfari, S. Lenzi

Abstract

This report presents the design and implementation of appropriate logical and physical data structures for Flash EEPROM memories, developed to increase access performance, reduce power consumption ad improve the device endurance. Such data structures have been exploited in the implementation of a driver for the physical data management layer of a lightweight DataBase Management System (DBMS), developed for devices with Flash memory storage. The DBMS has been designed from scratch to exploit the proposed logical and physical data structures, with the aim of supporting the use of context-aware applications, running on a portable device, working on a limited amount of data, which is opportunely selected based on the context.

I. Introduction

The large diffusion of portable consumer devices constituted by autonomous computational power, connectivity, limited battery life and storage – like cell phones, Portable Digital Assistants, or other devices such as the MIT 100$ laptop [11] – has led to the development of applications tailored for such kind of devices, and among them, DataBase Management Systems (DBMS) play a significant role. In fact, the resources of portable devices, although limited, allow the user to carry around a useful portion of data, to be read as well as modified. Such data may be a portion of a larger system (e.g., an employee's workload and tasks' data) or may be the unique copy of a user's information (e.g., personal internet access preferences and data): in both cases a portable DBMS is desirable as a back-end for accessing and managing data. The requested features are actually only a subset of the typical functionalities offered by traditional, full-size DBMS and applications; for instance, chances of having a concurrent access to data from different user applications on a portable device are limited. In this context, the query processing functionality providing the typical ACID properties plays a relevant role. On the other hand, it is important that the application be aware of the peculiar characteristics of the device, to better exploit the available resources without incurring in bottlenecks caused by the employed technology.

More precisely, these portable devices use Flash EEPROM memory as storage support and this kind of memory provides a particular read/write access, which significantly impacts on performance. Read and write operations can be performed with a bit/byte granularity, but data may be written only on previously erased locations, and erasure operations can be performed only with a block granularity. Such constraints affect both performance and power consumption, two aspects extremely important in battery-powered portable devices. Furthermore, Flash memory blocks can only be erased a finite number of times (up to 500,000) before becoming unreliable and thus unusable. As a result, when developing a data management application, particular attention should be devoted to the primitives accessing the storage medium.

In this scenario, we have developed specific logical and physical data structures for managing data to be stored on portable devices equipped with Flash EEPROM storage, by means of a portable light DBMS, dubbed *PoLiDBMS* [2]. The innovative contribution of this paper consists of an enhanced version of the data structures presented earlier ([3]), together with the design and implementation of a driver implementing the proposed policies. The initial proposed solutions have been re-designed to better exploit the evolving technological characteristics of Flash memories, as discussed in the next sections. These logical and physical data structures, together with the prototype PoLiDBMS are part of a more general methodology for Very Small DataBases (VSDB), aimed at supporting the designer in the design and management of limited context-aware data for mobile devices ([4]).

C. Bolchini, A. Giusti are with the Politecnico di Milano {bolchini,giusti}@elet.polimi.it
F. Furfari, S. Lenzi are with the CNR-ISTI {francesco.furfari, stefano.lenzi}@isti.cnr.it
This work has been partially supported by FIRB-MAIS project.

The rest of the paper is organized as follows. Section II presents an overview of other approaches related to either data management on portable devices, or Flash memory access. Section III presents the proposed logical and physical data structures constituting the lower level of the DBMS, implemented in a driver written in C and discussed in detail in Section IV. Future developments conclude the paper. Appendix A contains the source code related to the Sorted policy.

II. Related work

Nowadays the wide spread of portable devices has led to the development and use of several lightweight applications, among them there are also the DataBase Management Systems and several commercial producers have delivered scaled-down, reduced versions of well-known full-size DBMS, such as Oracle Database Lite – 10g [12], IAnywhere UltraLite Database [8], IBM DB2 Everyplace [9], and Microsoft SQL Server Mobile Edition [15]. An analysis of these systems highlighted a common factor among them: an underlying client-server architecture, where the portable device hosting the light DBMS is a client, and there is a full-featured server at the center of the architecture. Hence, the aim of these light DBMSes is to scale down an existing tool to fit on the reduced resources, providing a traditional database management system, built on top of the operating system of the portable device. As a result these applications do not have a specific control of read/write/erasure operations, and do not necessarily adopt ad-hoc data organization policies based on the identified peculiarities.

Nevertheless, when considering such a scenario and the limited resources, not all of the classical features of a DBMS are necessary, especially when considering the possibly reduced data held on the device and the fact that the SQL engine will serve a purpose of data access/manipulation rather than database creation, administration with possibly no direct user access. Furthermore, the particular technological characteristics of the storage medium suggest an accurate manipulation of the stored information to limit endurance degradation, power consumption and to achieve good performance. The proposed approach focuses the attention on these issues to design a light DBMS thought as a stand-alone system to provide the necessary data management features, but taking into account the particular technology hosting the system and the data.

An interesting survey on data structures and flash-specific file systems is presented in [6], [7], where the authors analyze several different approaches to exploit the peculiarities of Flash EEPROM memories as storage means. Most of the attention has been devoted, since the early 1990s, to the development of techniques and methods to provide memory usage that could emulate traditional storage (e.g., magnetic disks); some solutions for file system implementation are also reviewed, together with a few approaches aimed at allowing high-level software to access the flash device as a simple rewritable block device.

GnatDb is an embedded database system designed to run on a wide range of appliances, sometimes characterized by very limited resources and with relevant security issues [16]. The small footprint database system has been designed to reduce code footprint as well as stack and heap memory usage; furthermore the focus is on protecting data by means of security primitives integrated with log-sructured storage. No query processing is though provided and thus no investigation is carried out on necessary data structures except for the log-like one, which is suited for flash-memory-based storage.

PicoDBMS [13] is a database systems designed to be running on smart-cards, exceptionally limited-resource devices, able to execute query processing with no secrecy/security features. The authors propose an interesting data structure based on indexing to reduce the required memory space. The adoption of indices and their management is though not optimal for flash memories where erasures (to keep track of new index values) are both power- and performance-critical operations.

An open source DBMS is also available, sqlite [14], a small C library that implements a self-contained, embeddable, zero-configuration SQL database engine. Although fit for use in cellphones, PDAs, set-top boxes, and/or appliances, and usable as an embedded database in downloadable consumer applications, no specific features for low level data access and management are provided, using the operating systems primitive commands to read and write data.

Our contribution in this field consists in the design of appropriate data structures that can be used to for representing data to be accessed by means of a database management system, through traditional relational query processing, which exploit the technological characteristics of flash-memory storage. More precisely, we have focused the attention on defining an enhanced log-like storage management strategy, which is the most natural way for storing data on a Flash memory, suitable for tables in relational databases where data are only appended; moreover *heap* and *sorted* logical data structures have been also designed to "map" small unsorted and ordered data, respectively. Before introducing the proposed data structures, a few details on the Flash EEPROM technology are presented, highlight the motivations and peculiarities of the adopted solution; Section IV will then unveil the details of the implementation.

A. *Flash memory issues*

In general, two kinds of Flash memory implementations can be employed for portable device storage: NOR and NAND, the former usually adopted as file system storage, the latter for common consumer usage in the form of SmartMedia cards. Indeed NOR flash memories provides not much storage per chip, not being very dense, and are costly and slow to write; NAND flash are increasingly dense, low cost but have other limitations, so they are becoming more popular also for storage system implementations. Program and erasure operations require particular effort with respect to classical magnetic disk or RAM support, since a memory location needs to be erased before it can be programmed, independently of the granularity allowed by the specific type of memory. More in detail, write operations can only modify 0s to 1s, whereas changing 1s to 0s requires an erasure of the entire partition; moreover in NAND memories a page may be programmed a maximum number of times.

III. LOGICAL AND PHYSICAL DATA STRUCTURES

Classical, indexed data structures are often inappropriate for Very Small DataBases: the limited search needs we have within the relatively small tables we deal with is not worth the overhead required for managing and maintaining indexes; moreover, as we will show later, mantaining additional structures is very expensive with the present storage technology. Therefore we propose simple record-based tabular data structures, on which relational database operations can be easily mapped. Although these approaches have been preliminarily presented in [2], [3], during their implementation improvements have been introduced, and other interesting issues have also risen.

Logical data structures define how records are organized in the tables, whereas physical considerations allow ad-hoc optimizations tailored on the specific constraints of the flash memory substrate. In order to better understand the peculiarities and advantages of the proposed solutions, it is worth briefly analyzing the kind of operations that will be provided for accessing data. More precisely, the set of such operations we have envisioned and that are usually employed by the DBMS query processing engine to answer user's queries, are:

Scan returns all the records in the table;

Search returns only the records in the table which match a given condition; we support both equality ($\texttt{field} = \texttt{value}$) and range ($\texttt{minValue} \leq \texttt{field} \leq \texttt{maxValue}$) constraints;

Update updates a record in a table, modifying one or more fields;

Insert inserts a new record in a table;

Delete deletes a record;

In the following, a presentation of the logical organization of data is presented.

A. *Logical data structures*

We implemented 3 different logical data structures: each one has its advantages and drawbacks, which are mainly related to the relation's cardinality, and to the relative frequency of record insertion or update

operations w.r.t. search operations. Thus, at design time the database designer needs to perform an annotation task to characterize each table of the database with additional information to identify the most promising data structure according to the stored data and the estimated workload ([3]).

1) Heaps: Heap relations store records without any specific ordering. This is a simple, straightforward storage policy, useful if the relation's cardinality is limited, or if record searches are foreseen as extremely rare w.r.t. insert and update operations: infact, no overhead is imposed on operations which modify or add records. On the other hand, since records are not sorted, searches must always scan the entire heap of records instead of taking advantage of binary search algorithms.

2) Sorted tables: Sorted relations try to overcome these limitations: in a sorted relation, search operations can be completed in a very efficient manner if an equality or range condition is set on the sorting field.

However, when the storage substrate is a flash memory, mantaining a sorted relation when insert or update operations are issued is not easy: in the following, we will present the strategies we use in order to avoid a block erasure for each insert or update operation when a sorted table is involved.

3) Circular lists: A circular list is log-like relation with a fixed maximum number of records: when a new record is inserted, it is appended at the end of the existing list of records; if the maximum number of records is reached, the oldest one is erased. This is an example of how the DBMS includes additional knowledge related to the application/data being managed. Circular lists are usually sorted with respect to date and time.

B. Physical data structures

The peculiar characteristics of flash memories require ad-hoc optimizations in order to avoid an expensive block erasure operation when a record insertion, update or deletion is required.

1) deleted bit: Not all modifications of a flash memory block require a block erasure: in particular, bits in a block can always be set from `0` to `1` with a simple write operation; setting a bit in a block from `1` to `0`, on the other hand, always requires to erase the entire block.

Taking advantage of this behavior, we propose to reserve a single bit for each record, which we call *deleted bit*. A *deleted bit*, initially `0`, is set to `1` as soon as the associated record is deleted. The record deletion operation is then mapped to a simple write operation of the related deleted bit, which is extremely fast and inexpensive in terms of battery power. Scan and search operations simply ignore records with their *deleted bit* set.

When a block erasure operation is performed, we "garbage collect" records in the table which are marked as deleted, and permanently remove them.

This is an example of how performance of flash memory systems can be noticeably improved by avoiding expensive operations.

Deleted bits require just one bit per record: this is a neglegible overhead even for the smallest realistic record length: their use is therefore suggested even where delete operations are not very common. Obviously, deleted bits are useless in scenarios where records are never deleted nor updated.

2) distributed dummy records: Mantaining a sorted relation when record insertions and updates are involved, while avoiding block erasures is not straightforward. For example, if a new record must be inserted between two adjacent records, a block erasure is often unavoidable.

We propose to counter this issue by interleaving actual records with dummy, unprogrammed records which, being entirely composed by `0` bits, can be conveniently overwritten with a simple write operation. Dummy records are differentiated w.r.t. ordinary programmed records by means of a *programmed bit*, which is handled in the same way as the *deleted bit* we previously introduced.

The first step for inserting a new record is determining its position in the sorted table: if the new record can be inserted in place of a dummy record, the record insertion operation does not need a block erasure. On the contrary, if the records preceding and following the new one are adjacent and there is no available dummy record, a block erasure takes place; in this occasion, deleted records are garbage collected, and dummy records are redistributed.

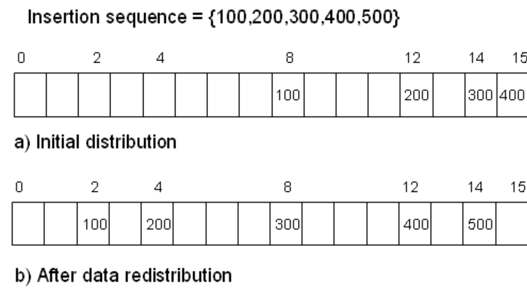

Fig. 1. Top: computed positions for worst case. Bottom: after record redistribution.

The efficiency of this strategy is highly dependent on the distribution of the dummy records in the table; in particular, we are evaluating the effectiveness of two strategies for record placement in the table, the first one an incremental improvement over the proposal in [3], the second one being an innovative version.

Hashing-function-driven strategy: Each record is associated with an "ideal" position in the table by means of a hashing function, which reflects the expected distribution of the records. When an insert operation is initiated, the ideal position of the new record is computed, and it is used as a base for its placement. If sorting can not be mantained, a block reprogramming is needed.

Since in most real-world usage scenarios the distribution of the records is not uniform, a good selection of the hashing function is of prominent importance. The hashing function should map the peculiar value distribution of the records' sorting field to a uniform distribution throughout the entire table, so that dummy records keep evenly distributed; this maximizes the probability that a dummy record is available for overwriting upon insertion of a new record.

If a sufficiently large example of the records the table has to deal with is available, the hashing function could be determined offline and automatically from the inferred value distribution of the sorting field; else, the VSDB designer should compose an appropriate function incorporating its knowledge of the application scenario.

Binary-tree-based strategy: The objective of this strategy is to achieve maximum record scattering also when a conflict arises because the ideal position is already occupied. In order to find the position that maximizes space between records the main idea is to use a binary-search like algorithm.

An insert operation consists in a binary search that looks for a record with the same ordering key; when the search ends we either (see figure 1):

- hit an empty location (dummy record), where we will write the record to be inserted: note that this placement will leave the maximum space between the new record and the adjacent records, already stored.
- find a record with the same key value, possibly immediately followed by other records with the same key: thus we can append the new record to the group, if space is available; else, a record redistribution is needed.

In figure 1, as an example, we report the computed positions for the worst case that occurs when the sequence of data to be inserted is already ordered. It is straightforward to notice that after $\log_2(N)$ insertions, where N is the logical size of the table, the new record should ideally be placed among the leaves of a tree whose depth is too high to be serialized on the table. In the figure the element that causes the fault is "500". In such situation, and others in which similar conflicts arise, we erase the block and proceed to reorder the sorted data starting from the middle element and applying recursively the algorithm to the left and right partition. In figure 1, bottom, we represent the element displacement after a block erasure, where the initial middle element is "300".

The algorithm just described has a couple of issues.

- If we delete a record the algorithm would not be able to find the right position to insert (search) a new record anymore; this problem is easily circumvented, leveraging on the logical deletion policy we introduced previously: a deleted record, which is still there but has the *deleted bit* set, can still be used by the comparison step of the binary search. The record is not needed anymore after a block erasure because of the reordering of the elements, so it can be safely garbage-collected.
- Since usually a table spans on more than one block, in order to maintain the elements ordered we have to erase many blocks for each data reordering: we are currently investigating the possibility to erase just the block where the conflict occurs.
  In our current implementation, we apply the algorithm to each block independently; the cost for partitioning the table into different blocks of sorted data does not increase the global computational cost when the number of blocks k is not comparable to $\log_2(N)$. We can populate each block gradually putting all the incoming data in the first block and moving to the second one when a conflict arises or, alternatively, distributing them among all the blocks in round robin fashion. The computational cost for this solution for the operations introduced in the Section III-A is: $O(N)$ for the scan operation considering we have to merge ordered lists of data coming from the different blocks, $O(log(n))$ for search (executed for each block), and $O(\log(N))$ for insert, delete and update (considered as a delete and insert combination) operations.

This strategy does not assume any knowledge of the distribution of the key values in the dataset, and automatically adapts itself by means of the incremental record redistribution process.

3) Metadata mantainance: The different logical and physical structures need to mantain some metadata, such as the number of records in the table and the position of the first record in a circular relation. However, mantaining updated metadata could introduce a significant overhead, therefore a number of strategies have been adopted:

a session based mechanism: metadata are kept in RAM until the end of the session, when they are written to flash. While efficient, this approach is dangerous because if the device is abruptly powered down before the end of the session, the tables' state is inconsistent;

ad-hoc optimizations for writing metadata: metadata consist of a dynamic list, where only the last of the inserted items is the "current" one. By means of smart overwriting, we manage to avoid block erasures most of the times, and keep an always-updated record of the current metadata.

IV. A PROTOTYPE DBMS IMPLEMENTATION

A DBMS for portable devices has been implemented ([2]) and is currently being re-engineered, providing data management and processing features. The query processing engine is written in Java: this allows rapid prototyping, and enables us to focus on algorithmic issues in a well-engineered modular system. The underlying driver, which implements logical and physical structures on the flash memory, is being implemented in C: we devoted much more attention to optimization, and provide a well-documented interface which can be used in other projects not needing query processing functionality. We use the familiar Linux distribution on the iPaq platform, since Linux allows easy low level interfacing with the flash memory; more precisely the used platform is the following one: HP5400 and HP3900 ipaqs equipped with a familiar linux distribution (0.7.2) [1] and a blackdown JVM version 1.3.1 [5].

A. Data Access Layer

A simplified view of the DBMS architecture is shown in figure 2. We focus on the Data Access Layer (DAL), which provides access to the flash memory and implements an abstract layer to allow different data storage implementations to be used. We used such mechanism during the initial testing of the entire DBMS in order to use an alternative XML based storage driver.

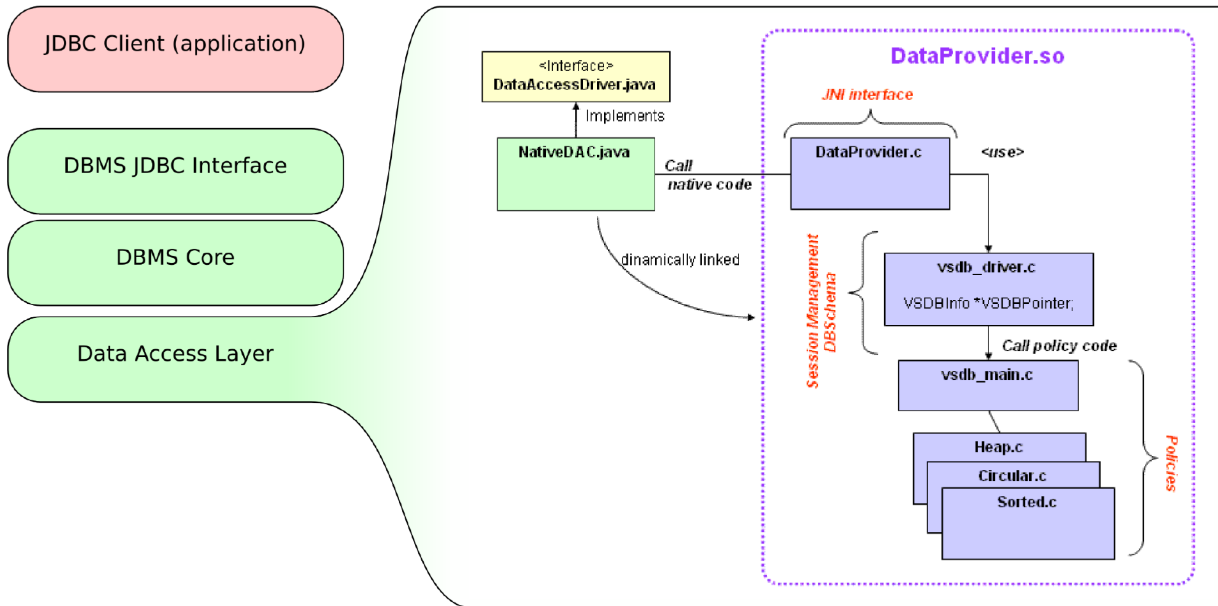

Fig. 2. The DBMS high-level architecture (left), and the detail on the Data Access Layer (right).

To provide a new data access implementation a component is required to implement the DataAccess-Driver interface that groups the five operations we introduced in Section III-A, along with methods for initializing and closing the session.

Figure 2 depicts the driver architecture. The Java class NativeDAC is the wrapper of the C code implementing the Flash Memory Driver. The driver is released as shared library (`DataProvider.so`) dynamically linked to JVM by the NativeDAC class. `DataProvider.c` is a proxy class that carries out of all the interfacing tasks between Java and C code by means of the JNI interface ([10]), while the real driver' entry point is represented by the `vsdb_driver.c` class. It instantiates the physical data structures needed to maintain the DB metadata and dispatches the invoked operations to the specific classes implementing the various logical data structures we introduced previously.

We have had to face problems related to the poor performance we obtained using JNI callbacks for data types conversion between Java and C code. This is especially true when considering that the not all the current JVMs always implement native code optimizations ([10]). Hence we decided to pass all the parameters as a single array of bytes (`Byte[]`) in every native invocation, instead of require many JNI callbacks. Thus, with a minimal cost due to some conversion routines in the Java code, we were able to retrieve the whole record stored in the flash memory and to use it directly in Java code.

We still need to solve several issues: so far the simple returning of a record implies copying of the data twice. The first time, using the Memory Technology Devices (MTD) subsystem, we copy the record from the flash memory to a local C data structure and the second one to move the data beyond the C boundaries by means of a JNI call to the `SetByteArrayRegion()` utility, that allows to copy the data into the internal data structures of the JVM. The copy of the data impacts especially on the scan operation, so we foresee to define new JNI-malloc and JNI-write functions as Java callbacks. They will be used to copy the data from the flash memory to the memory allocated directly by Java code. These functions will be part of a more abstract interface used by the basic I/O level routines in order to achieve a better portability of the C code. In this way we are not tied to Java code and also other C based client applications will be able to link and embed the VSDB driver without significant performance loss.

V. Conclusions and Future Work

The paper presents an enhanced set of logical and physical data structures for EEPROM Flash memories, aimed at improving the performance in data access and management on this kind of memory support.

Indeed, the peculiarity of the technological device suggests the definition and adoption of specific data structures, to minimize the limitation constituted by the constraint of erasing memory locations before re-writing them.

An entire methodology for designing and managing small amounts of data on portable devices has been developed, based on the definition of appropriate logical and physical data structures. These proposed solutions have been implemented and are currently being re-engineered and optimized in the lower layer of a portable light DBMS (PoLiDBMS) to be hosted on mobile devices equipped with Flash EEPROM memories.

VI. Acknowledgments

We would like to thank our colleagues Proff. F. Rabitti, F. A. Schreiber and L. Tanca for the useful discussions.

References

[1] Java-linux. `http://www.blackdown.org/`.
[2] C. Bolchini, C. Curino, M. Giorgetta, A. Giusti, A. Miele, F. A. Schreiber, and L. Tanca. Polidbms: Design and prototype implementation of a dbms for portable devices. In *Proc. of the 12th Italian Symposium on Advanced Database Systems (SEBD)*, pages 166–177, S. Margherita di Pula (CA) Italy, June 2004.
[3] C. Bolchini, F. Salice, F. A. Schreiber, and L. Tanca. Logical and physical design issues for smart card databases. *ACM Trans. Information Systems*, 21(3):254–285, 2003.
[4] C. Bolchini, F. A. Schreiber, and L. Tanca. A methodology for very small database design. *Information Systems*, to appear.
[5] Handhelds.org. `http://familiar.handhelds.org/`.
[6] E. Gal and S. Toledo. Algorithms and data structures for flash memories. *ACM Computing Survey*, 37(2):138–163, 2005.
[7] E. Gal and S. Toledo. Mapping structures for flash memories: Techniques and open problems. In *Proc. of the IEEE Int. Conf. on Software - Science, Technology & Engineering (SWSTE)*, pages 83–92, Washington, DC, USA, 2005. IEEE Computer Society.
[8] UltraLita Database by IAnywhere. Sybase document. `http://www.ianywhere.com/products/mobile.html`.
[9] DB2 Everyplace by IBM. IBM document. `http://www-306.ibm.com/soft--ware/data/db2/everyplace/index.html`.
[10] S. Liang. *Java(TM) Native Interface: Programmer's Guide and Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
[11] One laptop per child project. `http://laptop.org/`.
[12] Oracle Database Lite - 10g. Oracle document. `http://www.oracle.com/technology/products/lite/lite_datasheet_10g.pdf`.
[13] P. Pucheral, L. Bouganim, P. Valduriez, and C. Bobineau. Picodbms: Scaling down database techniques for the smartcard. *VLDB Journal*, 10(2-3):120–132, 2001.
[14] Sqlite. `http://www.sqlite.org/`.
[15] Microsoft SQL Server Mobile Edition. Microsoft document. `http://www.microsoft.com/sql/ce/productinfo/SQLMobile.asp`.
[16] R. Vingralek. Gnatdb: A small-footprint, secure database system. In *Proc. of the 28th Int. Conf. on Very Large Data Bases (VLDB)*, pages 884–893, Hong Kong, China, August 2002.

```c
/*
 * $Revision: 0.1 $
 * $Date: 2003/10/01 12:29:59 $
 * $Author montesissa
 *
 * $Revision: 2.0.0 $
 * $Date: 2006/01/13 12:00:00 $
 * $Author: Bruno Bacci (bruno@bacci)
 * $Author: Francesco Furfari (francesco.furfari@isti.cnr.it)
 * $Author: Stefano "Kismet" Lenzi (kismet-sl@users.sourceforge.net)
 *
 * Copyright (C) 2003 Politecnico di Milano
 * Dipartimento di Elettronica e Informazione - Milano (Italy)
 * All rights reserved.
 *
 * vsdb_dummy_sorted.c: Distributed Dummy Records Implementation for Sorted Logi
cal!
 *
 *
 *
 *
 * NOT YER COMPILED WITH GCC NOR TESTED ON ARM AND X86!!
 *
 */


#include <errno.h>
#include <string.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <linux/mtd/mtd.h>
#include "vsdb_define.h"
#include "vsdb_metainfo.h"
#include "vsdb_ipaq.h"
#include "vsdb_basic.h"
#include "vsdb_dummy_sorted.h"
#include "vsdb_dummy_heap.h"
#include <stdio.h>
#include <stdlib.h>

/**
 * This module contains all the function required to manage a table
 * implementing an sorted relation.
 * More later.......
 */


/*****************************************************************************
*************/
/*                              UTILITY FUNCTION                            */
/*****************************************************************************
*************/

/**
 * This function return the first PROGRAMMED and not DELETED Record inside the T
able that have
 * index Equal or Greater/Lesser(depend of the value speciefied by Direction) th
an the RecordNumber
 *
 * @param Flash: is the flash device we are working on (thah is the MTD storing
the VSDB database).
 * @param DBHandle is the data structure maintaining all the metadata informatio
n.
 * @param Table is the pointer to the TableInfo data structure containing the me
tadata inforrmation
 *               related to the table from which we want extract a record.
 * @param RecordNumber is the position of the record inside the table represente
d by TableInfo
 *              that will used as starting position.
 * @param RecordBuffer preallocated memory area used to store the record read fr
om the table.
 * @param UpToIndex is the limit to use for searching for a valid Record
 * @param Direction if SEARCH_LEFT will search for valid record that has index t
hat is Equal or Greater,
 *                  if SEARCH_RIGHT will search for valid record that has index th
at is Equal or Lesser
 *
 * @return the index of the found record, UpToIndex+1 if no record are found, or
 negative value on error
 *
 * @note RecordBuffer must be pre allocated with dimension Table->RowSize
 * @note No free will be call by this function
 */
inline int GetNextValidRecord(int Flash, struct VSDBInfo *DBHandle, struct Table
Info *Table,
      long RecordNumber, void *RecordBuffer, long UpToIndex, short Direction)
{
   int ret;

   while(RecordNumber≤UpToIndex){
        if ((ret = GetRecordFromTable(Flash, DBHandle, Table, RecordNumber, Recor
dBuffer)) ≠ VSDB_SUCCESSFUL) { /* Error reading a record */
           return ret;
        }
       if(GetRecordStatus(RecordBuffer, Table→RowSize) ≡ VSDB_PROGRAMMED)
           return RecordNumber;

       RecordNumber+=Direction;
   }

   return RecordNumber;
}
/**
 * Use the binary search to find the location near to value to search.
 * Near because it return alway a index even if the value searched is not
 * present.
 *
 * The function assume that:
 *  - If during a binary search I find a empty location that one should be
 *      the one perfect for the insert so every record with lower index have low
er
 *      value and every record with higher index have higher value
 *  - If I find the record that exactly the position that I was looking for savi
ng I have
 *      only to check for some island of the same value
 *  - If I don't find the value it means that last pivot is the value nearest
 *
 *
 * @param Flash
 * @param DBHandle
 * @param Table
 * @param ValueToMatch
 * @param Start
 * @param End
 * @param Direction //TODO Remove useless
 *
 * @return the position near to the given value
 *
 */
long SearchNearestPositionSorted(int Flash, struct VSDBInfo *DBHandle, struct Ta
bleInfo *Table,
      void *ValueToMatch, int Start, int End, short Direction)
{
   int Last,First,Middle,ret;
   unsigned char  *RecordBuffer;

   /*
    * Makes room for the buffer used to read a record from the table.
    */
   if ((RecordBuffer = (unsigned char *)malloc(Table→RowSize*Table→UsedBlocks)
) ≡ NULL) /* Failed to allocate memory */
       return VSDB_NOT_ENOUGH_MEM;

   Last=End;
   First=Start;
   while(Start ≤ End ){
```

vsdb_dummy_sorted.c

```c
      /* Starting binary like research */
      Middle=(Start + End) / 2;

      if( (ret=GetRecordFromTable(Flash, DBHandle, Table, Middle, RecordBuffer))
 ≠ VSDB_SUCCESSFUL) return ret;
      switch(GetRecordStatus(RecordBuffer,Table→RowSize)){
         case VSDB_EMPTY:{
            /* Return the position of the non-empty record, based on the option
of the caller */
            if(Direction ≡ SEARCH_RIGHT) return End;
            else return Start;
         }break;
         case VSDB_DELETED: case VSDB_PROGRAMMED:{
            switch (CompareKeyRecords(ValueToMatch,RecordBuffer,Table)){
               case -1: { /* We have to select the upper(left) part of the block
 */
                  End = Middle - 1;
               } break;
               case 1:{ /* We have to select the lower(right) part of the block
*/
                  Start = Middle + 1;
               } break;
               case 0:{
                  return Middle;
               }break;
            }
         }break;
      }
   }
   return Middle;
}

/**
 * This function is used to get data related to one of the field of the Record
 *
 * @param Record pointer to buffer that contain a record
 * @param Field pointer to metadata related to field that we want get the value
 *
 * @return pointer (within the buffer given)to value of specified Field
 */
inline unsigned char * GetSortedData ( unsigned char * Record, struct Field * Fi
eld)
{
   return (void*) (Record + Field→offset);
}

/**
 * This function is used to compare two Record on the sorting key
 * @param ra pointer to first record
 * @param rb pointer to second record
 * @param Table   pointer to metadata related to Table that contain those record
.
 *
 * @return   0 if the kay of the record have same value,
 *          -1 if the key of ra is lower than the key of rb
 *           1 if the key of ra is greater than the key of rb
 *
 */
inline int CompareKeyRecords(void * ra, void * rb, struct TableInfo * Table)
{
   struct Field * MatchField;
   void *raKey, *rbKey;
   MatchField=Table→ExtendedInfo.Sorted.SortedField;

   raKey=GetSortedData(ra,MatchField);
   rbKey=GetSortedData(rb,MatchField);

   return SearchCompare(raKey,rbKey,rbKey,MatchField→domain);
}

/**
 *
 * Is implementaion of the algorithm to rewrite data on the just erased bloack s
o that it will leave the maximum space within block.
 * The idea of the algorithm is:
 * Because the MegaBuffer is an ordered vector we can read it as an BinarySearch
Tree, and the Toot of the Tree is the "mediana" of the
 * whole vector, and the Root of each SubTree is the moda applied to vector that
 conatin all the element of that Tree.
 * For Example consider the following data:
 *
 * *---*---*---*---*---*---*---*---*---*---*---*---*
 * | 2 | 5 | 7 | 12| 31| 45| 51| 55| 65| 66| 70| 80|
 * *---*---*---*---*---*---*---*---*---*---*---*---*
 *
 * you can see the above that as the following tree
 *                        *---*
 *                           | 45|
 *                           *---*
 *                *---*        |          *---*
 *                | 7 |<------------------>| 65|
 *                *---*                 *---*
 *         *---*       |        *---*  *---*      |       *---*
 *         | 2 |<----------->| 12|  | 51|<---------->| 70|
 *         *---*              *---*  *---*              *---*
 *            |    *---* *---*    |      |    *---**---*    |      *---*
 *            --->| 5 | | 31|<---        --->| 55|| 66|<------->| 80|
 *                 *---* *---*                *---**---*          *---*
 *
 *     and the above view of the vector can be built by applying the algorithm.

 * The algorithm act as follow:
 *
 * @param Flash is the file descriptor of the MTD device. The device need to be
 *         open befor this function is called.
 * @param DBHandle is the data structure holding metadata information.
 * @param Table Metadata of the table that is used for reading and storing opera
tion
 * @param BlockStart Index of the first Record of the Block involeved in Garbagi
ng
 * @param BlockEnd Index of the last Record of the Block involeved in Garbaging
 * @param MegaBuffer Pointer to the buffer storing all the Record that must rest
ore on inside the Table
 * @param Start Starting point of the MegaBuffer to use
 * @param End End point of the MegaBuffer to use
 *
 * @return VSDB_SUCCESSFUL if and only if there is no error and all the record a
re been written back.
 *          On error may be possible that part of the block is now used
 *
 */
int GarbageSortedDummyOnWrite(
   int Flash, struct VSDBInfo * DBHandle, struct TableInfo *Table,
   int BlockStart, int BlockEnd, unsigned char * MegaBuffer, int Start, int End
)
{
   int Middle,Left,Right,MiddleBlock, ret;

   if(Start>End) return VSDB_SUCCESSFUL;

   /* FIX Critical Error Data losses */
   if((BlockStart<0)∨(BlockStart>BlockEnd)) return VSDB_SORTED_NOSPACE;

   /* Root of the Tree of starting at Start and ending at End */
   Middle=(Start+End)/2;
   Left=Middle-1;
   Right=Middle+1;

   /* Middle of the Block */
   MiddleBlock=(BlockStart+BlockEnd)/2;

   if( (ret=WriteRecordToTable(Flash,DBHandle,Table,MiddleBlock,&(MegaBuffer[Mid
dle]))) ≠ VSDB_SUCCESSFUL) return ret;
   MiddleBlock--;
   /* Iterating on the Left SubTree*/
   if( (ret=GarbageSortedDummyOnWrite(Flash,DBHandle,Table,BlockStart,MiddleBloc
k,MegaBuffer,Start,Left)) ≠ VSDB_SUCCESSFUL) return ret;

   MiddleBlock=(BlockStart+BlockEnd)/2;
   MiddleBlock++;
```

```c
    /* Iterating on the Right SubTree*/
    if( (ret=GarbageSortedDummyOnWrite(Flash,DBHandle,Table,MiddleBlock,BlockEnd,
MegaBuffer,Right,End)) ≠ VSDB_SUCCESSFUL) return ret;

    return VSDB_SUCCESSFUL;
}


/**
 *
 * @param Flash
 * @param DBHandle
 * @param Table
 * @param MetaInfo
 * @param Block
 * @param NewRecord
 *
 * @return VSDB_SUCCESSFUL if and only if no error are occurred.
 */
int GarbageSortedDummy(
    int Flash, struct VSDBInfo * DBHandle, struct TableInfo *Table,
    struct SortedTableInfo *MetaInfo, int Block, void *NewRecord
)

{
     int ValidRecords, i, ret,EndRecord,StartRecord;
     unsigned char *MegaBuffer, *RecordBuffer;
     struct FlashBlock *CurrentBlock;

     /*
      * The RecordBuffer is used to store a generic record read from
      * the table. If the record is valid thet it will be copied
      * into the MegaBuffer.
      */
     if ((RecordBuffer = (unsigned char *)malloc(Table→RowSize)) ≡ NULL)
    return VSDB_NOT_ENOUGH_MEM;

     /*
      * This buffer holds valid record only, that is
      * record that are not marked as deleted.
      */
     if ( (MegaBuffer = (unsigned char *)malloc((MetaInfo→Blocks[Block].ValidRec
ord+1) * (Table→RowSize) )) ≡ NULL){
       free(RecordBuffer);
       return VSDB_NOT_ENOUGH_MEM;
     }

     /*
      * Select from the table only valid records.
      * We scan all the table (and thus all the blocks to
      * find all the valid records.
      */
     ValidRecords = 0; /* Initialize the number of records that we must write bac
k */
     (MetaInfo→Blocks[Block].ValidRecord)++; /* I'll consider even the new Recor
d that must be inserted */
     for(i=0; i<MetaInfo→Blocks[Block].ValidRecord; i++) {
    /*
     * Read the record in the position "i"
     */
    if ((ret = GetRecordFromTable(Flash, DBHandle, Table, i, RecordBuffer)) ≠ VSD
B_SUCCESSFUL) { /* Something wrong reading the i-th record */
        free(RecordBuffer);
        free(MegaBuffer);
        return ret;
    }
    if( GetRecordStatus(RecordBuffer, Table→RowSize) ≡ VSDB_PROGRAMMED) { /* The
 record is programmed and not deleted so we save it */
        if(CompareKeyRecords(NewRecord,RecordBuffer,Table)≥0){
            memcpy((void *)(MegaBuffer+(ValidRecords*Table→RowSize)),NewRecord,Ta
ble→RowSize);
            ValidRecords++; /* Update the number of valid record. */
            break;
        }
        memcpy((void *)(MegaBuffer+(ValidRecords*Table→RowSize)),RecordBuffer,Ta
ble→RowSize);
        ValidRecords++; /* Update the number of valid record. */
    }
     }
     /* I'll add the remaining Record after the insertion of the NewRecord */
     for(; i<MetaInfo→Blocks[Block].ValidRecord; i++) {
    /*
     * Read the record in the position "i"
     */
    if ((ret = GetRecordFromTable(Flash, DBHandle, Table, i, RecordBuffer)) ≠ VSD
B_SUCCESSFUL) { /* Something wrong reading the i-th record */
        free(RecordBuffer);
        free(MegaBuffer);
        return ret;
    }
    if( GetRecordStatus(RecordBuffer, Table→RowSize) ≡ VSDB_PROGRAMMED) { /* The
 record is programmed and not deleted so we save it */
        memcpy((void *)(MegaBuffer+(ValidRecords*Table→RowSize)),RecordBuffer,Ta
ble→RowSize);
        ValidRecords++; /* Update the number of valid record. */
    }
     }
     /*
      * Erase block used to store the table data.
      */
     CurrentBlock=&(Table→Blocks[Block]);
    if((ret = EraseBlock(Flash, CurrentBlock)) ≡ -1) { /* .....erase it */
        free(RecordBuffer);
        free(MegaBuffer);
        return ret;
    }

     /*
      * STARTING WRITING BACK
      *
      * Write back the valid records collected into the
      * MegaBuffer buffer.
      */

    /*Finding the first record index within the block*/
    StartRecord = Table→NumberRecordPerBlock * Block;

    /*Finding the last record index within the block*/
    EndRecord = Table→NumberRecordPerBlock * Block + Table→NumberRecordPerBlock
 - 1;

     if ((ret = GarbageSortedDummyOnWrite(Flash,DBHandle,Table,StartRecord,EndRec
ord,MegaBuffer,0,ValidRecords)) ≠ VSDB_SUCCESSFUL){
        free(MegaBuffer);
        free(RecordBuffer);
        return ret;
     }
        free(MegaBuffer);
        free(RecordBuffer);

     /*
      * We need to update the metadata information.
      * In this case the number of valid, programmed and used
      * record is equal to ValidRecord.
      */
     (Table→ValidRecords)++;
     (Table→UsedRecords)++;
     Table→ProgrammedRecords = Table→ProgrammedRecords - MetaInfo→Blocks[Block]
.ProgrammedRecord + ValidRecords;
     MetaInfo→Blocks[Block].ProgrammedRecord = ValidRecords;
     MetaInfo→Blocks[Block].ValidRecord = ValidRecords;

     /*
      * This will be used when we decide to
      * add support to journaling.
      */
     /*AddDynamicCommand(DBHandle,Table->Number, CMD_CHANGE_NPR, Table->Programme
dRecords,0);
       AddDynamicCommand(DBHandle,Table->Number, CMD_CHANGE_NVR, Table->ValidReco
rds,0);
       AddDynamicCommand(DBHandle,Table->Number, CMD_CHANGE_NUR, Table->UsedRecor
ds,0);
       */
       return VSDB_SUCCESSFUL;
}


/**
 * This function is used to find the sorting Field of a Table
 *
 * @param Table a struct TableInfo * of which we want to know which Field is use
d as sorting key
 * @return  a struct Field * that point to the Field of the given Table that is
used as sorting key
 */
struct Field *  GetSortedField(struct TableInfo * Table)
{
    /*
     * Used static field to be able to cache the order field used for the table
     */
    static int i = -1;
    struct Field * SortedField = 0;

    if(SortedField≢0)
       return SortedField;

    for(i=0;i<Table→FieldsNumber; i++){
       if(Table→Fields[i].sorted≡1){
          SortedField=&(Table→Fields[i]);
          return SortedField;
       }
    }

    return 0;
}

/**
 * This function is in charge of inserting a new record on an heap table.
 * If there is not room for the record but there are some record marked as
 * deleted, a garbage collection is performed. If record marcked as
 * deleted are not found, the insertion fails.
 *
 * @param Flash is the file descriptor of the MTD device. The device need to be
 *        open befor this function is called.
 * @param DBHandle is the data structure holding metadata information.
 * @param Table is the position of the table we are working on inside the array
Tables
 *        mantained in the DBHandle data structure.
 * @param MetaInfo pointer to metadata that we are using
 * @param Block is the block to use for the insertion
 * @param NewRecord is the new record we must insert inside the table defined by
 TableNumber
 * @param RecordBuffer is the new buffer that can store even status information
of the NewRecord
 *
 * @return  >0 if and only if the the record is been inserted and the number mea
ns to location where
 *          the record is been stored. Any lesser than 0 value means error
 *
 * @note No check is done for identify the Table to be a SortedTable
 * @note This method do not free any memory localtion and do not allocate any me
mory location
 */
inline int InsertSortedDummyOnBlock(
    int Flash, struct VSDBInfo *DBHandle, struct TableInfo *Table,
    struct SortedTableInfo *MetaInfo, int Block,
    void *NewRecord, void * RecordBuffer){

    int StartRecord,EndRecord,MiddleRecord,InsertPosition,ret;
    unsigned char * BaseAddr;

    /* InserPositio will cotain the position of the record or -1 if no free space
 are available -2 means not initialized */
    InsertPosition=-2;

    /*Finding the first record index within the block*/
    StartRecord = Table→NumberRecordPerBlock * Block;

    /*Finding the last record index within the block*/
    EndRecord = Table→NumberRecordPerBlock * Block + Table→NumberRecordPerBlock
 - 1;

    while(StartRecord ≤ EndRecord ∧ InsertPosition ≡ -2){
       /* Starting binary like research */
       MiddleRecord=(StartRecord + EndRecord) / 2;

       if( (ret=GetRecordFromTable(Flash, DBHandle, Table, MiddleRecord, RecordBu
ffer)) ≠ VSDB_SUCCESSFUL) return ret;
       switch(GetRecordStatus(RecordBuffer,Table→RowSize)){
          case VSDB_EMPTY:{ /* If free space found we use that record as destinat
ion */
             InsertPosition=MiddleRecord;
          }break;
          case VSDB_DELETED: case VSDB_PROGRAMMED:{ /* If non-free space we itera
te on half of block */
             switch (CompareKeyRecords(NewRecord,RecordBuffer,Table)){
                case -1: case 0:{ /* We have to select the upper(left) part of th
e block */
                   EndRecord = MiddleRecord - 1;
                } break;
                case 1:{ /* We have to select the lower(right) part of the block
*/
                   StartRecord = MiddleRecord + 1;
                } break;
             }
          }break;
       }
    }

    if(InsertPosition>-1){
        /*
         * Copies the record that must be inserted in the RecordBuffer and add th
e
         * bytes satting that the record is programmed. Finally write the record
         * in to the flash.
         */
        memcpy(RecordBuffer, NewRecord, Table→RowSize - 2);
        BaseAddr = (unsigned char *)RecordBuffer + (Table→RowSize -2);
        *BaseAddr = VSDB_TRUE_BYTE;
        if ((ret=WriteRecordToTable(Flash, DBHandle, Table, InsertPosition, Recor
dBuffer)) ≠ VSDB_SUCCESSFUL) return ret;

        return InsertPosition;
    }else{
       return VSDB_SORTED_NOSPACE;
    }
}

/*****************************************************************************
************/
/*                              STORING FUNCTION                          */
/*****************************************************************************
************/




/**
 * This function is in charge of inserting a new record on an heap table.
 * If there is not room for the record but there are some record marked as
 * deleted, a garbage collection is performed. If record marcked as
 * deleted are not found, the insertion fails.
 *
 * @param Flash is the file descriptor of the MTD device. The device need to be
 *        open befor this function is called.
 * @param DBHandle is the data structure holding metadata information.
 * @param TableNumber is the position of the table we are working on inside the
array Tables
 *        mantained in the DBHandle data structure.
 * @param NewRecord is the new record we must insert inside the table defined by
```

```c
  TableNumber
  *
  * @note No check is done for identify the Table to be a Sorted
  */
int InsertSortedDummy(int Flash, struct VSDBInfo *DBHandle, unsigned int TableNu
mber, void *NewRecord)
{
    int ret,i;
    struct TableInfo *Table;
    unsigned char  *RecordBuffer;
    struct SortedTableInfo *MetaInfo;
    unsigned long MaxRecords;

    /*
     * Find the metadata information related to the table
     * having position TableNumber inside the array of table
     * mantained in the DBHandle data structure.
     */
    Table = &(DBHandle→Tables[TableNumber]);
    MetaInfo = &(Table→ExtendedInfo.Sorted);

    /*
     * Computes the maximum number of record that can be stored
     * inside the table, including the deleted one.
     */
    MaxRecords = Table→NumberRecordPerBlock * Table→UsedBlocks;

     /*
      * If the number of valid record is equal to the maximum
      * number of records that can be stored inside the table
      * we return a table full and the operation fails.
      */
    if(Table→ValidRecords ≡ MaxRecords)
   return VSDB_TABLE_FULL;

    /*
     * There is room to store a new record so
     * we can perform the operation. First of all
     * we allocate memory for a temporary record.
     */
    if( (RecordBuffer = (unsigned char *)malloc(Table→RowSize)) ≡ NULL)
      return VSDB_NOT_ENOUGH_MEM;

    /*
     * There are three possible situation:
     * 1) There is free space on the block and the correct position on the curre
nt block is free => we have just to insert the record
     * 2) There is no space for the correct position on the current block => we
try to insert on the next block
     * 3) There is no space but all the block do not have the correct position f
ree => garbaging one of the block and try to insert there
     */
    for(i=0;i<4;i++){
      if( MetaInfo→Blocks[MetaInfo→CurrentBlock].ProgrammedRecord < Table→Numb
erRecordPerBlock ) { /* I'll if is not full by ProgrammedRecord*/
          if( (ret=InsertSortedDummyOnBlock(Flash,DBHandle,Table,MetaInfo,MetaInf
o→CurrentBlock,NewRecord,RecordBuffer)) ≥ 0 ) break;
        }
        MetaInfo→CurrentBlock = (MetaInfo→CurrentBlock + 1) % Table→UsedBlocks;
    }

   if(ret≡VSDB_SORTED_NOSPACE){/* All the block do not have an hole for the inse
rtation, we have to do garbaging and trying to insert agin */
      for(i=0;i<4;i++){
          if( MetaInfo→Blocks[MetaInfo→CurrentBlock].ValidRecord < Table→Number
RecordPerBlock ) { /* I'll if is not full by ValidRecord */
          if( (ret=GarbageSortedDummy(Flash,DBHandle,Table,MetaInfo,MetaInfo→Cur
rentBlock,NewRecord)) ≡ VSDB_SUCCESSFUL ) {
              break;
          }}
           MetaInfo→CurrentBlock = (MetaInfo→CurrentBlock + 1) % Table→UsedBloc
ks;
      }

      /* Regardless that record is been inserted we have to free memory */
      free(RecordBuffer);
      return ret;
    }


   if(ret≥0){ /* Means that I have inserted the record so I have to updated meta
data */
       (Table→ValidRecords)++;
       (Table→UsedRecords)++;
       (Table→ProgrammedRecords)++;

       /* We change the next record to insert, so we can do interleaving on orde
r stream that is the worst situation for this algorithm*/
      (MetaInfo→Blocks[MetaInfo→CurrentBlock].ProgrammedRecord)++;
      (MetaInfo→Blocks[MetaInfo→CurrentBlock].ValidRecord)++;
      MetaInfo→CurrentBlock = (MetaInfo→CurrentBlock + 1) % Table→UsedBlocks;

      /* TODO Every X insertion we should select the less used block as next ins
ertion block */

      ret=VSDB_SUCCESSFUL;
    }

    /*
     * We will use these commd when we implement the
     * journaling support.
     */
    /* AddDynamicCommand(DBHandle,TableNumber, CMD_CHANGE_NPR, Table->Programmed
Records,0);
    AddDynamicCommand(DBHandle,TableNumber, CMD_CHANGE_NUR, Table->UsedRecords,0
);
    return AddDynamicCommand(DBHandle,TableNumber, CMD_CHANGE_NVR, Table->ValidR
ecords,0); */

   /* Regardless that record is been inserted we have to free memory */
   free(RecordBuffer);
   return ret;


}

/**
 * This function is in charge of retriving from an ordered list of record, from
 * a all the blocks of the table were record are stored between StartPosition[b]
 * and EndPosition[b] where b is the block index
 *
 * The algorithm used is a standard multi list merge sort. Where we use a ValidB
locks
 * array to store information related to the status of each head of the list:
 * -1 head reached the tail, no more record to read
 *  0 head is valid and can be used
 *  1 head is not valid because was been used, but we can search for next elemen
t
 * The algorithm have 2 phase:
 * 1 - Till there is 2 or more head valid
 *     1.1 - Init all the head
 *   1.2 - Find the minium of the head and attach to the result buffer
 * 2 - If there is one head valid read and copy the value of all the valid recor
d
 *           to the result buffer
 *
 * @params Flash is the file descriptor related to the VSDB we are working on.
 * @param DBHandle is the pointer to metadata information describing the VSDB we
 are working on.
 * @param TableNumber is the information that allows to find the table where get
 record.
 * @param StartPositions is an array of length equal to the number of block of t
he table
 *        and contain information related which is the first position of the reco
rd that
 *        that we have to inspect
 * @param EndPositions is an array of length equal to the number of block of the
 table
 *        and contain information related which is the last position of the recor
d that
 *        we have to inspect
 * @param OutputBuffer a buffer big enough to store all the result. A safe value
can be
 *        ValidRecord metadata of the used table.
 *
 * @return number of readen record on success or <0 on error;
 */
int GetSortedOutput(int Flash, struct VSDBInfo *DBHandle, unsigned char TableNum
ber,
   long *StartPositions,long *EndPositions, void * OutputBuffer)
{

   struct TableInfo *Table;
   long ValidRecords, ret;
   unsigned char  *RecordBuffer, *OutputCurrent,*CurrentBuffer,*MinimumRecord;
   int i,*ValidBlocks,minimum,NumValidBlocks;

   /*
    * First of all gets all the information describing
    * the table we are working on.
    */
   Table = &(DBHandle→Tables[TableNumber]);

   /*
    * Makes room for the buffer used to read a record from the table.
    */
   if ((RecordBuffer = (unsigned char *)malloc(Table→RowSize*Table→UsedBlocks)
) ≡ NULL) /* Failed to allocate memory */
       return VSDB_NOT_ENOUGH_MEM;

   /*
    * Makes room for vector that indicate which block are still usable and if we
 need to read more record from them
    */
   if ((ValidBlocks = (int *)malloc(sizeof(int)*Table→UsedBlocks)) ≡ NULL){ /*
Failed to allocate memory */
      free(RecordBuffer);
      return VSDB_NOT_ENOUGH_MEM;
   }else{
      /* Initializing the vector to all the first record of each block */
      /*
       * ValidBlocks contains the following value:
       * -1 => No more valid record can be retrived from the block with same ind
ex
       *  0 => A record is been read from the block and can be readed from Recor
dBuffer
       *  1 => The record inside RecordBuffer that was read now can be overwritt
en because it was used
       */
      NumValidBlocks=0;
      for(i=0;i<Table→UsedBlocks;i++){
         /* Only blocks where START is less equal than END are valid */
         if(StartPositions[i] ≤ EndPositions[i]){
            ValidBlocks[i] = 1;
            NumValidBlocks++;
         }else{
            ValidBlocks[i] = -1;
         }

      }

   }

   /*
    * Start a loop where we scan all the record using the merge sort algorithm w
ithin
    * the used blocks of the table
    */
   ValidRecords = 0;
   OutputCurrent=OutputBuffer;

   /* PHASE 1 */
   while(NumValidBlocks > 1  ∧ ValidRecords < Table→ValidRecords) {
      /* PAHSE 1.1 Init all the head - Reading all the RecordBuffer that are con
sumed */
      for(i=0;i<Table→UsedBlocks;i++){
         /* Means not consumed or reach end of block */
         if(ValidBlocks[i]≤0) continue;
         /*TODO Read only record that are been consumed */
         /*??? It's better calculate CurrentBuffer on-demand or keep it up-to-da
te and increase it every time we increase i */
         CurrentBuffer = RecordBuffer + (i * Table→RowSize);
         if ((ret = GetNextValidRecord(Flash, DBHandle, Table, StartPositions[i]
, (void *) CurrentBuffer,EndPositions[i],SEARCH_RIGHT)) < 0){
            free(RecordBuffer);
            free(ValidBlocks);
            return ret;
         }else if(ret ≡ (EndPositions[i] + 1) ){
            /**
             * No valid record found we set the block as ended
             */
            ValidBlocks[i]=-1;
            NumValidBlocks--;
            /* The insertion of the last block will be performed on next cycle *
/
            if( NumValidBlocks ≡ 1 ) break;
         }else{
            /*
             * We update the star position because we have just read from block
             * tha may mean that StartPositions[i] now is greater than EndPositi
ons[i]
             * if we read the last available record from block.
             */
            StartPositions[i] = ret+1;
            ValidBlocks[i]=0;
         }
      }
      /*
       * PHASE 1.2
       *
       * minimum index is preserved as the previus
       * There is always at least a ValidBlocks because otherwise it means that
I should be exited at
       * the previous loop iteration because ValidRecords == Table->ValidRecords
       *
       * We usa loop to find which Blocks contain the valid record.
       */

      for(i=0;i<Table→UsedBlocks;i++){
         if(ValidBlocks[i]<0) continue;
         minimum=i;
         break;
      }
      /* Finding minum of the key */
      MinimumRecord=RecordBuffer + (minimum * Table→RowSize);
      CurrentBuffer=RecordBuffer;
      for(i=0;i<Table→UsedBlocks;i++){
         /* Skip non valid record and avoid comparison between "me and me"*/
         if(ValidBlocks[i]<0 ∨ minimum≡i){
            CurrentBuffer += Table→RowSize;
            continue;
         }
         if(CompareKeyRecords(MinimumRecord,CurrentBuffer,Table)≡1){
            minimum = i;
            MinimumRecord = CurrentBuffer;
         }
         CurrentBuffer += Table→RowSize;
      }

       /*
        * Now we copy to record to the output buffer
        */
      memcpy(OutputCurrent, MinimumRecord, Table→RowSize-2);
      OutputCurrent += (Table→RowSize-2);
      ValidRecords++;
      /* Set record as consumed */
      ValidBlocks[minimum]=1;
   }

   /**
```

```c
     * PHASE 2
     *
     * Reading all the reamin record from only block that is no more empty
     * So we assume that only one ValidBlocks is available or none are available
     */
    CurrentBuffer=RecordBuffer;
    for(i=0;i<Table→UsedBlocks;i++){
       if(ValidBlocks[i]<0) continue;
       minimum=i;
       break;
    }
    while(NumValidBlocks ≡ 1  ∧ ValidRecords < Table→ValidRecords) {
       if ((ret = GetNextValidRecord(Flash, DBHandle, Table, StartPositions[minim
um], (void *) CurrentBuffer,EndPositions[minimum],SEARCH_RIGHT)) < 0){
          free(RecordBuffer);
          free(ValidBlocks);
          return ret;
       }else if(ret ≡ (EndPositions[i] + 1) ){
          /**
           * No valid record found we set the block as ended
           */
          NumValidBlocks--;
       }else{
          /*
           * We update the star position because we have just read from block
           * tha may mean that StartPositions[i] now is greater than EndPositions
[i]
           * if we read the last available record from block.
           */
          StartPositions[minimum] = ret+1;
          memcpy(OutputCurrent, CurrentBuffer, Table→RowSize-2);
          OutputCurrent += (Table→RowSize-2);
          ValidRecords++;
       }
    }
    free(RecordBuffer);
    free(ValidBlocks);
    return ValidRecords;
}

/**
 * This function is in charge of load all the valid records (i.e record that are
 * programmmed but not delete) in the memory.
 * It return the records in the same order were they are store on the physical l
ayer.
 *
 * @param Flash is the file descriptor we are working on
 * @param DBHandle is the data structure holding all the metadata information
 * @param TableNumber the table number in the array of tables
 * @param Records the scanned records are stored in tyhis variable (not allocate
d from the caller).
 * @return the number of valid record loaded in to Record on succees, an integer
< 0 otherwise.
 */
int ScanRawSortedDummy(int Flash, struct VSDBInfo *DBHandle, unsigned char Table
Number, void **Records)
{
    /*
     * Because there is no differnce between how record are stored inside the phy
sical layer
     * and how they are read with the Scan we can use the default Scan
     */
    return ScanHeapDummy(Flash, DBHandle, TableNumber, Records);
}

/**
 * This function is in charge of retriving all the valid record stored in a circ
ular relation.
 *
 * @params Flash is the file descriptor related to the VSDB we are working on.
 * @param DBHandle is the pointer to metadata information describing the VSDB we
 are working on.
 * @param TableNumber is the information that allows to find the table where we
will insert
 *         a new record.
 * @param Records is the buffer where all the valid records will be stored.
 *
 * @return number of readen record on success or <0 on error;
 */
int ScanSortedDummy(int Flash, struct VSDBInfo *DBHandle, unsigned char TableNum
ber, void ** Records)
{
    struct TableInfo *Table;
    long *StartPositions,*EndPositions, ValidRecords;
    unsigned char  *OutputBuffer;
    int i;

    /*
     * First of all gets all the information describing
     * the table we are working on.
     */
    Table = &(DBHandle→Tables[TableNumber]);

    /*
     * We know that there are ValidRecords records
     * stored in to the tables. Note: we do not return
     * to the caller delete and programmed byte.
     */
    if((OutputBuffer = (unsigned char *)malloc((Table→RowSize - 2) * (Table→Val
idRecords))) ≡ NULL) { /* Error allocating memory */
       return VSDB_NOT_ENOUGH_MEM;
    }

    /*
     * Makes room for vector that point to head of each block;
     */
    if ((StartPositions = (long *)malloc(sizeof(long)*Table→UsedBlocks)) ≡ NULL)
{ /* Failed to allocate memory */
       free(OutputBuffer);
        return VSDB_NOT_ENOUGH_MEM;
    }else{
       /* Initializing the vector to all the first record of each block */
       /* TODO We can use a better start position based on the number of
        * available records that is BlockStart + ( NumberRecordForBlock / ( 2^Use
dRecordOfThisBlock ) )
        */
       for(i=0;i<Table→UsedBlocks;i++){
          StartPositions[i]=Table→NumberRecordPerBlock * i;
       }
    }

    /*
     * Makes room for vector that point to end of each block;
     */
    if ((EndPositions = (long *)malloc(sizeof(long)*Table→UsedBlocks)) ≡ NULL){
/* Failed to allocate memory */
       free(OutputBuffer);
       free(StartPositions);
        return VSDB_NOT_ENOUGH_MEM;
    }else{
       /* Initializing the vector to all the first record of each block */
       /* TODO We can use a better end position based on the number of
        * available records that is BlockEnd - ( NumberRecordForBlock / ( 2^UsedR
ecordOfThisBlock ) )
        */
       for(i=0;i<Table→UsedBlocks;i++){
          EndPositions[i]=Table→NumberRecordPerBlock * (i+1) -1;
       }
    }

    ValidRecords=GetSortedOutput(Flash,DBHandle,TableNumber,StartPositions,EndPos
itions,OutputBuffer);

    if(ValidRecords<0) free(OutputBuffer);
    else *Records=(void*)OutputBuffer;
    free(StartPositions);
    free(EndPositions);

    return ValidRecords;
}
```

vsdb_dummy_sorted.c

```c
/**
 * This function is in charge of erasing a record from the table.
 * The erasing operation is a logical one, that is the record is marked
 * as deleted and will be physically erased when a garbage operation is
 * performed.
 *
 * @param Flash is the file descriptor of the MTD device. The device need to be
 *         open befor this function is called.
 * @param DBHandle is the data structure holding metadata information.
 * @param TableNumber is the position of the table we are working on inside the
array Tables
 *         mantained in the DBHandle data structure.
 * @param RecordToDelete is the position of the record inside the table. It is w
hort noticing
 *         that this number is relative to the valid record, that is record marke
d as deleted
 *         are not considered. Said in another way, if RecordToDelete is 10, then
we must count 10 valid
 *         record, without considering the deleted records.
 *
 * @return VSDB_SUCCESSFUL if and only if there was no error occur and a record
is been deleted,
 *        VSDB_NO_RECORD_FOUND if and only if no error happend but no record are
been deleted
 *
 */
inline int DeleteSortedDummy(int Flash, struct VSDBInfo *DBHandle, unsigned char
 TableNumber, long RecordToDelete)
{
    struct TableInfo *Table;
    long CurrentRecord,ValidRecords,nur,Position,Start,End;
    int ret,block;
    unsigned char  *RecordBuffer;

    /*
     * Get the table metadata we need to delete
     * the record.
     */
    Table = &(DBHandle→Tables[TableNumber]);

    /*
     * Allocate temporary memory to store the record
     * we read from the flash. In fact we need to load a record
     * and check if it is programmed or not. If is programmed than
     * we consider to reach the record to delete (that is we increment
     * the ValidRecord variable), else we ignore it.
     */
    if((RecordBuffer = (unsigned char *)malloc(Table→RowSize)) ≡ NULL)
   return VSDB_NOT_ENOUGH_MEM;

    /*
     * This was a nasty bug as we supose that the user
     * start to count index from 0 so also ValidRecords
     * mus strat from -1. The original code follows.
     * ValidRecords = 0;
     * CurrentRecord = -1;
     */
    ValidRecords = -1;
    CurrentRecord = -1;
    nur = Table→UsedRecords;
    Start = 0;
    End = Table→NumberRecordPerBlock * Table→UsedBlocks - 1;
    while(ValidRecords ≠ RecordToDelete) {/* Scans the reord until we found Reor
dToDelete valid records */
       Position=GetNextValidRecord(Flash,DBHandle,Table,Start,(void *)RecordBuffe
r,End,SEARCH_RIGHT);
       if(Position<0) /* Error found */
          return Start;
       if(Position>End) /* No more record available inside the Table */
          break;
       Start=Position+1;
       ValidRecords++;
    }

    /*
     * If we found RecordToDelete valid records then
     * CurrentRecord is the position of the record we want delete.
     * Otherwise we didn't found the record.
     */
    if(ValidRecords ≠ RecordToDelete) {
   free(RecordBuffer);
   return VSDB_NO_RECORD_FOUND;
    }

    /*
     * Set the delete byte to signal that the record is
     * programmed but not valid.
     */
    if((ret=SetDeletedByte(Flash, DBHandle, Table, Position)) ≠ VSDB_SUCCESSFUL)
 /* Something wrong deleting the record */
   return ret;

    /*
     * Update the number of valid record (that is
     * record that are programmed but not deleted).
     */
    (Table→ValidRecords)--;
    /*I've only to updated the specialized MetaData information*/
    block=RecordToDelete / DBHandle→Tables[TableNumber].NumberRecordPerBlock;
    (DBHandle→Tables[TableNumber].ExtendedInfo.Sorted.Blocks[block].ValidRecord
)--;

    /*
     * We will use command when we will implement
     * the journaling.
     */
    /* AddDynamicCommand(DBHandle,TableNumber, CMD_CHANGE_NVR, Table->ValidRecor
ds,0); */

    free(RecordBuffer);
    return VSDB_SUCCESSFUL;
}

/**
 * This function is in charge of change one or more filed of a record.
 *
 * @param Flash the file descriptor where the VSDB is sitored.
 * @param DBHandle is the data structure holding all the metadata information.
 * @param TableNumber the table number in the array of tables.
 * @param RecordToUpdate is the order number of the recrd to be updated.
 * @param UpdatedRecord is the new record to be inseted (takes the place of the
previous one).
 *
 * @return VSDB_SUCCESSFUL if and only if there was no error
 *
 */
int UpdateSortedDummy(int Flash, struct VSDBInfo *DBHandle, unsigned char TableN
umber, long RecordToUpdate, void *UpdatedRecord)
{
    int ret;

    if ((ret = DeleteSortedDummy(Flash, DBHandle, TableNumber, RecordToUpdate))
≠ VSDB_SUCCESSFUL)
   return ret;

    return InsertSortedDummy(Flash, DBHandle, TableNumber, UpdatedRecord);
}

/**
 * This function is in charge of retriving all the records having in the field i
dentified by MatchFieldNumber
 * a velue included in the interval identified by ValueMinToMatch and ValueMaxTo
Match.
 * The resuls are stored in the buffer Records.
 *
 * @param Flash is the file descriptor of the flash partition where we are worki
ng on.
 * @param DBHandle is the pointer to the metadata information related to the VSD
B we are working on.
```

```
 * @param TableNumber idenifies the table wher we are executing the operation.
1055  * @param MatchFieldNumber identifies the field used to discriminate if a field
will be included in the computed result.
 * @param ValueMinToMatch is the left value of the interval.
 * @param ValueMaxToMatch is the right value of the interval.
 * @param Records holds all the records that match the condition.
 *
1060  * @return on success the number of record found, <0 on error
 */
int SearchRangeSortedDummy(int Flash, struct VSDBInfo *DBHandle, unsigned char T
ableNumber, unsigned int MatchFieldNumber,
              void *ValueMinToMatch,void *ValueMaxToMatch, void **Records){



1065
   struct TableInfo *Table;
   long *StartPositions,*EndPositions, ValidRecords,s,e;
   unsigned char *OutputBuffer;
   short i;
1070   struct SortedTableInfo *MetaInfo;

    /*
     * Find the metadata information related to the table
     * having position TableNumber inside the array of table
1075     * mantained in the DBHandle data structure.
     */
    Table = &(DBHandle→Tables[TableNumber]);
    MetaInfo = &(Table→ExtendedInfo.Sorted);

1080   if(MatchFieldNumber≠MetaInfo→SortedFieldIndex){
      return SearchRangeHeapDummy(Flash,DBHandle,TableNumber,MatchFieldNumber,Va
lueMinToMatch,ValueMaxToMatch,Records);
   }

   /*
1085    * We know that there are ValidRecords records
    * stored in to the tables. Note: we do not return
    * to the caller delete and programmed byte.
    */
   if((OutputBuffer = (unsigned char *)malloc((Table→RowSize - 2) * (Table→Val
idRecords))) ≡ NULL) { /* Error allocating memory */
1090      return VSDB_NOT_ENOUGH_MEM;
   }

   /*
    * Makes room for vector that point to head of each block;
1095    */
   if ((StartPositions = (long *)malloc(sizeof(long)*Table→UsedBlocks)) ≡ NULL)
{ /* Failed to allocate memory */
      free(OutputBuffer);
       return VSDB_NOT_ENOUGH_MEM;
   }else{
1100      /* Initializing the vector to all the first record of each block */
      for(i=0;i<Table→UsedBlocks;i++){
         StartPositions[i]=Table→NumberRecordPerBlock * i;
      }
   }
1105
   /*
    * Makes room for vector that point to end of each block;
    */
   if ((EndPositions = (long *)malloc(sizeof(long)*Table→UsedBlocks)) ≡ NULL){
/* Failed to allocate memory */
1110      free(OutputBuffer);
      free(StartPositions);
       return VSDB_NOT_ENOUGH_MEM;
   }else{
      /* Initializing the vector to all the first record of each block */
1115      for(i=0;i<Table→UsedBlocks;i++){
         EndPositions[i]=Table→NumberRecordPerBlock * (i+1) -1;
      }
   }

1120   /**
    * Refeing the start and end by searching min and max value with the followin
g idea:
    * if we are searching for all X where A<=X<=B but inside our table do not co
ntain one
    * of exteme value, let's assume A is missing so we search all X wher A'<=X<=
B
    * where A' is the minimum value contained in the table greater than A.
1125    */
   for(i=0;i<Table→UsedBlocks;i++){
      s=StartPositions[i];
      e=EndPositions[i];
      /*
1130       * Leverange the previus example now we search for A' of each block
       * So we use the SEARCH_RIGHT option
       */
      StartPositions[i]=SearchNearestPositionSorted(Flash, DBHandle, Table, Valu
eMinToMatch, s, e, SEARCH_RIGHT);
      /*
1135       * Leverange the previus example now we search for B' of each block
       * Please note that this time we are looking for the maximum value
       * contained inside the table lesser than B so we use SEARCH_LEFT option
       */
      EndPositions[i]=SearchNearestPositionSorted(Flash, DBHandle, Table, ValueM
axToMatch, s, e, SEARCH_LEFT);
1140   }

   ValidRecords=GetSortedOutput(Flash,DBHandle,TableNumber,StartPositions,EndPos
itions,OutputBuffer);

   if(ValidRecords≤0) free(OutputBuffer); /* <0 means error, =0 means no record
found */
1145   else *Records=(void*)OutputBuffer;
   free(StartPositions);
   free(EndPositions);

   return ValidRecords;
1150
}
```