# Database Pointers: a Predictable Way of Manipulating Hot Data in Hard Real-Time Systems*

Dag Nyström[1], Aleksandra Tešanović[2],
Christer Norström[1], and Jörgen Hansson[2]

[1] Dept. of Computer Engineering, Mälardalen University
{dag.nystrom,christer.norstrom}@mdh.se
[2] Dept. of Computer Science, Linköping University
{alete,jorha}@ida.liu.se

**Abstract.** Traditionally, control systems use ad hoc techniques such as shared internal data structures, to store control data. However, due to the increasing data volume in control systems, these internal data structures become increasingly difficult to maintain. A real-time database management system can provide an efficient and uniform way to structure and access data. However the drawback with database management systems is the overhead added when accessing data. In this paper we introduce a new concept called database pointers, which provides fast and deterministic accesses to data in hard real-time database management systems compared to traditional database management systems. The concept is especially beneficial for hard real-time control systems where many control tasks each use few data elements at high frequencies. Database pointers can co-reside with a relational data model, and any updates made from the database pointer interface are immediately visible from the relational view. We show the efficiency with our approach by comparing it to tuple identifiers and relational processing.

## 1 Introduction

In recent years, the complexity of embedded real-time controlling systems has increased. This is especially true for the automotive industry [1]. Along with this increased complexity, the amount of data that needs to be handled has grown in a similar fashion. Since data in real-time systems traditionally is handled using ad hoc techniques and internal data structures, this increase of data is imposing problems when it comes to maintenance and development.

One possible solution to these problems is to integrate an embedded real-time database management system (RTDBMS) within the real-time system. A RTDBMS can provide the real-time system with a uniform view and access of data. This is especially useful for distributed real-time systems where data is

* This work is supported by ARTES, a network for real-time research and graduate education in Sweden.

shared between nodes. Because of the uniform access of data, the same database request is issued regardless if the data is read at the local node or from a distributed node. Furthermore, RTDBMSs can ensure consistency, both logical and temporal [2]. Finally, RTDBMSs allow so called ad hoc queries, i.e., requests for a view of data performed during run-time. This is especially useful for management and system monitoring. For example, consider a large control system being monitored from a control room. Suddenly, a temperature warning is issued. An ad hoc query showing the temperatures and pressures of multiple sub-systems might help the engineers to determine the cause of the overheating.

Integrating a RTDBMS into a real-time system also has drawbacks. There will most certainly be an added overhead for retrieving data elements. This is partly because of the indexing system used by most database management systems (DBMS). The indexing system is used to locate where in the memory a certain data element is stored. Usually, indexing systems use some tree structure, such as the B-tree [3] and T-tree [4] structures, or a hashing table [5].

An increase of the retrieval times for data has, apart from longer task execution, one additional drawback. Since shared data in a concurrent system needs to be protected using semaphores or database locking systems, the blocking factor for hot data can be significant. Hot data are data elements used frequently by multiple tasks. Hot data is sensitive to congestion and therefore it is of utmost importance to lock hot data for as short time as possible. Furthermore, it is important to bound blocking times to allow response time analysis of the system. Examples of hot data are sensor readings for motor control of a vehicle, e.g., rpm and piston position. These readings are continuously stored by I/O tasks and continuously read by controlling tasks. A congestion involving these heavily accessed data elements might result in a malfunction. On the other hand, information regarding the level in the fuel tank is not as crucial and might be accessed less frequent, and can therefore be considered non-hot data.

In this paper we propose the concept of database pointers, which is an extension to the widely used tuple identifiers [6]. Tuple identifiers contain information about the location of a tuple, typically a block number and an offset. Database pointers have the efficiency of a shared variable combined with the advantages of using a RTDBMS. They allow a fast and predictable way of accessing data in a database without the need of consulting the DBMS indexing system. Furthermore database pointers provide an interface that uses a "pointer-like" syntax. This interface is suitable for control system applications using numerous small tasks running at high frequencies. Database pointers allow fast and predictable accesses of data without violating neither temporal or logical consistency nor transaction serialization. It can be used together with the relational data model without risking a violation of the database integrity.

The paper is outlined as follows. In section 2 we describe the type of systems we are focusing on. In addition, we give a short overview of tuple identifiers and other related work. Database pointers are explained in section 3, followed by an evaluation of the concept, which is presented in section 4. In section 5 we conclude the paper.

## 2 Background and Related Work

This paper focuses on real-time applications that are used to control a process, e.g., critical control functions in a vehicle such as motor control and brake control. The flow of execution in such a system is: (i) periodic scanning of sensors, (ii) execution of control algorithms such as a PID-regulators, and (iii) propagation of the result to the actuators.

The execution is divided into a number of tasks, e.g., I/O-tasks and control tasks. The functions of these tasks are fixed and often limited to a specific activity. For example, an I/O-task's only responsibility could be to read the sensor-value on an input-port and write it to a specific location in memory, e.g., a shared variable [7].

In addition to these, relatively fixed control tasks, a number of management tasks exists, which are generally more flexible than the control tasks, e.g., management tasks responsible for the user interface.

### 2.1 Relational Query Processing

Relational query processing is performed using a data manipulation language (DML), such as SQL. A relational DML provides a flexible way of viewing and manipulating data. The backside of this flexibility is performance loss.

Figure 1 shows a typical architecture of a DBMS. The DBMS provides access to data through the SQL interface. A query, requesting value $x$, passed to this interface will go through the following steps:

1. The query is passed from the application to the SQL interface.
2. The SQL interface requests that the query should be scheduled by the transaction scheduler.
3. The relational query processor parses the query and creates an execution plan.
4. The locks needed to process the query are obtained by the concurrency controller.
5. The tuple containing $x$ is located by the index manager.
6. The tuple is then fetched from the database.
7. All locks are released by the concurrency controller.
8. The result is returned to the application.

Finally, since the result from a query issued to a relational DBMS is a relation in itself, a retrieval of the data element $x$ from the resulting relation is necessary. This is done by the application.

In this example we assume a pessimistic concurrency control policy. However, the flow of execution will be roughly the same if a different policy is used.

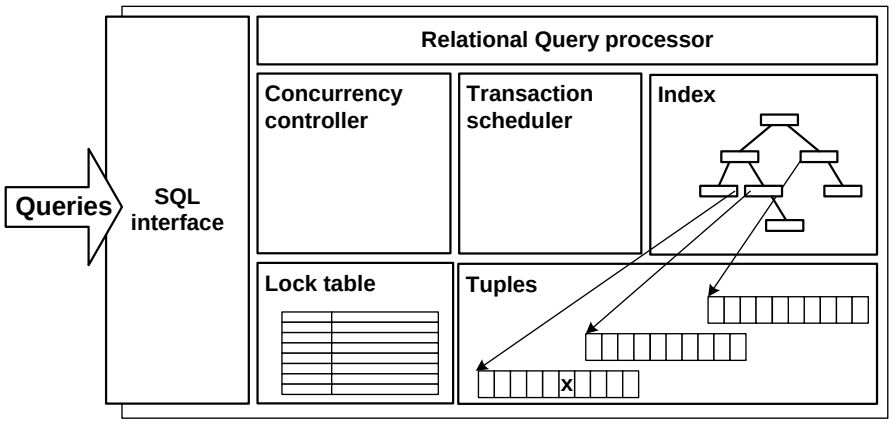


**Fig. 1.** Architecture of a typical Database Management System.

### 2.2 Tuple Identifiers

The concept of tuple identifiers was first proposed back in the 70's as internal mechanisms for achieving fast accesses to data while performing relational operations, such as joins and unions. It was implemented by IBM in an experimental prototype database called System R [6]. A tuple identifier is a data type containing a pointer to one tuple stored either on a hard drive or in main memory. Usually, a tuple is a rather short array of bytes containing some data. For a relational model, one tuple contains the data for one row of a relation.

A decade later, it was proposed in [8] that tuple identifiers could be used directly from the application via the DBMS interface. This would enable applications to create shortcuts to hot data, in order to retrieve them faster. The concept is also implemented in the Adabas relational DBMS [9] under the name *Adabas Direct Access Method*. In Adabas, tuple identifiers are stored in a hash table and can be retrieved by the user for direct data access. A disadvantage of this concept is the inability to move or delete tuples at run-time. To be able to perform deletions or movements of tuples in Adabas, a reorganization utility must be run, during which the entire database is blocked.

Applications using tuple identifiers must be aware of the structure of the data stored in the tuples, e.g., offsets to specific attributes in the tuple. This makes it difficult to add or remove attributes from relations, since this changes the structure of the tuples.

### 2.3 Related Work

Apart from tuple identifiers, the concept of bypassing the index system to achieve faster data access has been recognized in other database systems. The RDM database [10] uses a concept called network access, which consist of a network

of pointers. Network pointers shortcut data used in a predefined order. The implementation is, however, static and cannot be dynamically changed during run-time.

In the Berkeley database [11], a concept called queue access is implemented, which allows enqueueing and dequeueing of data elements without accessing the index manager. The approach is primarily suited for data production and consumption, e.g., state machines.

The Pervasive.SQL database [12], uses the interface Btrieve to efficiently access data. Btrieve supports both physical and logical accesses of tuples. Logical accesses uses a tuple key to search for a tuple using an index, while physical access retrieves tuples based on their fixed physical locations. One database file contains tuples of the same length in an array. Btrieve provides a number of operations that allows stepping between the tuples, e.g., `stepNext` or `stepLast`. The Btrieve access method is efficient for applications in which the order of accesses is predefined and the tuples are never moved during run-time. Furthermore, restructuring the data within the tuples is not possible.

Some database management systems use the concept of database cursors as a part of their embedded SQL interface [13]. Despite the syntactical similarities between database pointers and database cursors they represent fundamentally different concepts. While database cursors are used to access data elements from within query results, i.e., result-sets, database pointers are used to bypass the index system in order to make data accesses more efficient and deterministic.

## 3 Database Pointers

The concept of database pointers consists of four different components:

- The `DBPointer` data type, which is the actual pointer defined in the application.
- The database pointer table, which contains all information needed by the pointers.
- The database pointer interface, which provides a number of operations on the database pointer.
- The database pointer flag, which is used to ensure consistency in the database.

Using the concept of database pointers, the architecture of the DBMS given in figure 1, is modified to include database pointer components, as shown in figure 2. To illustrate the way database pointers work, and its benefits, we use the example presented in section 2.1, i.e., the request for retrieving the data $x$ from the database.

Using the database pointer interface, the request could be made significantly faster and more predictable. First, a read operation together with the database pointer would be submitted to the database pointer interface. The database pointer, acting as an index to the database pointer table array would then be used to get the corresponding database pointer table entry. Each database pointer table entry consists of three fields: the physical address of data element

$x$, information about the data type of $x$, and eventual locking information that shows which lock $x$ belongs to. Next the lock would be obtained and $x$ would be read. Finally, the lock would be released and the value of $x$ would be returned to the calling application. The four components of the database pointer and its operations are described in detail in sections 3.1 to 3.4.

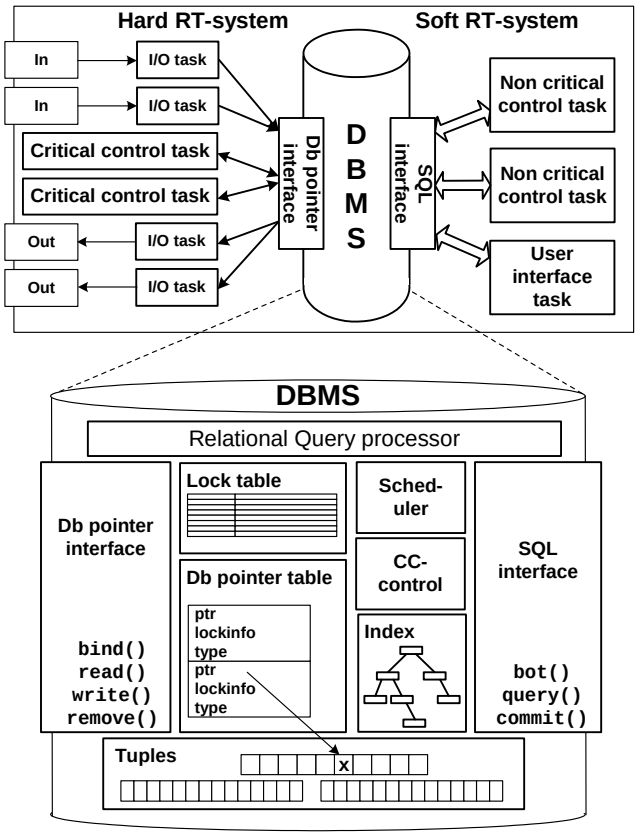


**Fig. 2.** Architecture of a controlling system that uses a DBMS with database pointers.

### 3.1 The `DBPointer` Data Type

The `DBPointer` data type is a pointer declared in the application task. When the pointer is initialized, it points to a database pointer table entry, which in its turn points to the actual data element. Hence the `DBPointer` could be viewed as a handle to a database pointer. However, due to the database pointer's syntactical similarities with a pointer variable, we have chosen to refer to it as a pointer.

### 3.2 The Database Pointer Table

The database pointer table contains all information needed for the database pointer, namely:

1. A pointer to the physical memory location of the data element inside the tuple. Typically, the information stored is the data block the tuple resides in, an offset to the tuple, and an offset to the data element within the tuple.
2. The data type of the data element pointed by the database pointer. This is necessary in order to ensure that any write to the data element matches its type, e.g., it is not feasible to write a floating point value to an integer.
3. Lock information describing the lock that corresponds to the tuple, i.e., if locking is done on relation granules, the name of the relation should be stored in as lock information. Note, if locks are not used in the DBMS, i.e., if optimistic concurrency control is used, some other serialization information can be stored in the database pointer table entry instead of the lock information.

### 3.3 The Database Pointer Interface

The database pointer interface consists of four operations:

1. `bind(ptr,q)` This operation initializes the database pointer *ptr* by binding it to a database pointer table entry, which in turn points to the physical address of the data. The physical binding is done via the execution of the query $q$, which is written using a logical data manipulation language, e.g., SQL. The query should be formulated in such a way that it always returns the address of a single data element. By using the bind operation, the binding of the data element to the database pointer is done using a logical query, even though the result of the binding is physical, i.e., the physical address is bound to the database pointer entry. This implies that no knowledge of the internal physical structures of the database is required by the application programmer.
2. `remove(ptr)` This operation deletes a database pointer table entry.
3. `read(ptr)` This operation returns the value of the data element pointed by *ptr*. It uses locking if necessary.
4. `write(ptr,v)` This operation writes the value $v$ to the data element pointed by *ptr*. It also uses locking if necessary. Furthermore, the type information in the database pointer entry is compared with the type of $v$ so that a correct type is written.

The pseudo codes for the `write` and `read` operations are shown in figure 3. The `write` operation first checks that the types of the new value matches the type of the data element (line 2), and then obtains a write lock for the corresponding lock (line 4), i.e., locks the relation that the data element resides in. The data element is then updated (line 5), and finally the lock is released (line 6). The `read` operation obtains the corresponding read lock (line 10), reads the data element (line 11), releases the lock (line 12), and then returns the value to the application (line 13).

```
1  write(DBPointer dbp, Data value){
2     if(DataTypeOf(value) != dbp->type)
3        return DATA_TYPE_MISMATCH;
4     DbGetWriteLock(dbp->lockInfo);
5     *(dbp->ptr) = value;
6     DbReleaseLock(dbp->lockInfo);
7     return TRUE;
   }

8  read(DBPointer dbp){
9     Data value;
10    DbGetReadLock(dbp->lockInfo);
11    value = *(dbp->ptr);
12    DbReleaseLock(dbp->lockInfo);
13    return value;
   }
```

**Fig. 3.** The pseudo codes for the `write` and `read` operations

### 3.4 The Database Pointer Flag

The database pointer flag solves the problem of inconsistencies between the index structure and the database pointer table, thus enabling tuples to be restructured and moved during run time.

For example, if an additional attribute is inserted into a relation, e.g., a column is added to a table, it would imply that all tuples belonging to the relation need to be restructured to contain the new data element (the new column). Hence, the size of the tuples changes, relocation of the tuples to new memory locations is most probable. Since a schema change is performed via the SQL interface, it will use and update the index in the index manager. If one of the affected tuples is also referenced from a database pointer entry, inconsistencies will occur, i.e., the database pointer entry will point to the old physical location of the tuple.

Each database pointer flag that is set in the index structure indicates that the tuple flagged is also referenced by a database pointer. This informs the index manager that if this tuple is altered, e.g., moved, deleted, or changed, the corresponding database table entry must be updated accordingly.

### 3.5 Application Example

To demonstrate how a real-time control system could use a RTDBMS with a database pointer interface, we provide an application example. Consider the system shown in figure 2 which is divided into two parts:

1. A hard real-time part that is performing time-critical controlling of the process. The tasks in this part use the database pointer interface.

2. A soft real-time part that handles user interaction and non-critical controlling. It uses the flexible SQL interface.

A hard real-time controlling task that reads a sensor connected to an I/O port is shown in figure 4. The task reads the current sensor value and updates the corresponding data element in the database. The task consists of two parts, an initialization part (line 2-4), which is run one time, and an infinite loop that is periodically polling the sensor and writing the value to the database (line 5-8).

The initialization of the database pointer is done by first declaring the database pointer (line 3) and then binding it to the data element containing the oil temperature in the engine (line 4). The actual binding is performed in the following four steps:

1. A new database pointer table entry is created.
2. The SQL query is executed and the address of the data element in the tuple is stored in the database pointer table entry.
3. The data type information is set to the appropriate type, e.g., `unsigned int`.
4. The locking information is set, e.g., if locking is done at relation granules, the locking information would be set to `engine`.

```
1 TASK OilTempReader(void){
2   int s;
3   DBPointer *ptr;
4   bind(&ptr, "SELECT temperature
              FROM engine WHERE
              subsystem=oil;");
5   while(1){
6     s=read_sensor();
7     write(ptr,s);
8     waitForNextPeriod();
    }
  }
```

engine

| subsystem | temperature | pressure |
|---|---|---|
| hydraulics | 42 | 27 |
| oil | 103 | 10 |
| cooling water | 82 | 3 |

**Fig. 4.** An I/O task that uses a database pointer and its corresponding relation.

After performing these four steps, the database pointer is initialized and ready to be used. The control loop is entered after the initialization (line 5). In the control loop a new sensor value is collected (line 6), the value is then written to the RTDBMS using the database pointer operation `write` (line 7). Finally, the task sleeps until the next period arrives (line 8).

## 4 Concept Evaluation

In table 1 we compare the different access methods: tuple identifiers (TiD's), database pointers (DbP's), and relational processing (Rel). Both tuple identifiers

| Criteria | | TiD's | DbP's | Rel |
|---|---|---|---|---|
| Interface | Pointer based | x | x | |
| | Relational | | | x |
| Data access | Physical | x | x | |
| | Logical | | x | x |
| Characteristics | Can handle tuple movements | | x | x |
| | Can handle attribute changes | | x | x |

**Table 1.** A comparison between tuple identifiers, database pointers, and relational processing.

and database pointers use a pointer based interface, which provides fast and predictable accesses to data inside a DBMS. However, it is not as flexible as most relational interfaces, e.g., SQL.

Furthermore, database pointer and tuple identifiers both access data based on direct physical references, in contrast to relational accesses that use logical indexing to locate data. However, database pointers bind the pointer to the data element using logical indexing, but access the data element using physical access.

Tuple identifiers have two drawbacks, firstly they are sensitive to schema changes, and secondly the physical structure of the database is propagated to the users. The former results in a system that can only add tuples instead of moving or deleting them, while the latter requires that the application programmer knows of the physical implementation of the database. Database pointers remove both of these drawbacks. Due to the flag in the index system, the database pointer table can be updated whenever the schema and/or index structure is changed, allowing attribute changes, tuple movements and deletions. Moreover, since the database pointer is bound directly to a data element inside the tuple instead of to the tuple itself, no internal structures are exposed.

The major advantage with accessing the data via pointers instead of going through the index system is the reduction of complexity. The complexity for the T-tree algorithm is $O(log_2 n + \frac{1}{2} log_2 \frac{n}{k})$, where $n$ is the number of tuples in the system and $k$ is the number of tuples per index node [14]. The complexity for database pointers and tuple identifier is $O(1)$. As can be seen, there is a constant execution time for accessing a data element using a database pointer or a tuple identifier, while a logarithmic relationship exists for the tree-based approach. There is however one additional cost for using the relational approach which we will illustrate with the following example.

We already showed how the oil temperature of an engine can be accessed using database pointers. Figure 5 shows the pseudo code for the same task, which now uses an SQL interface instead of the database pointer interface. In line 5, the `Begin of transaction` is issued and the actual update is performed in line 6, using a C-like syntax that resembles of the function `printf`. The actual `commit` is performed in line 7. In figure 5 all tuples in the relation `engine` have to be accessed to find all that fulfill the condition `subsystem = oil`. This requires accessing all three tuples.

```
1 TASK OilTempReader(void){
2   int s;
3   while(1){
4      s=read_sensor();
5      DB_BOT();
6      DB_Op("UPDATE engineSET temperature=%d
              WHERE subsystem = oil;",s);
7      DB_COMMIT();
8      waitForNextPeriod();
    }
  }
```

**Fig. 5.** An example of a I/O task that uses a Relational approach.

It can, of course, be argued that precompiled transactions would be used in a case like this. Precompiled transactions are transactions that have been evaluated and optimized pre-run time. Such transactions can be directly called upon during run-time, and is normally executed much more efficient than an ad-hoc query. However, this does not influence the number of tuples accessed, since no information of the values inside the tuples are stored there. Therefore, all three tuples have to be fetched anyway.

## 5 Conclusions and Future Work

In this paper we have introduced the concept of database pointers to bypass the indexing system in a real-time database. The functionality of a database pointer can be compared to the functionality of an ordinary pointer. Database pointers can dynamically be set to point at a specific data element in a tuple, which can then be read and written without violating the database consistency. For concurrent, pre-emptive applications, the database pointer mechanisms ensure proper locking on the data element.

We have also showed an example of a real-time control application using a database that supports both database pointers and a SQL interface. In this example the hard real-time control system uses database pointers, while the soft real-time management system utilizes the more flexible SQL interface.

The complexity of a database operation using a database pointer compared to a SQL query is significantly reduced. Furthermore, the response time of a database pointer operation is more predictable.

Currently we are implementing database pointers as a part of the COMET DBMS, our experimental database management system [15]. This implementation will be used to measure the performance improvement of database pointers for hard real-time controlling systems. Furthermore, different approaches for handling the interference between the hard real-time database pointer transactions and the soft real-time management transactions are investigated.

## References


1. Casparsson, L., Rajnak, A., Tindell, K., Malmberg, P.: Volcano - a revolution in on-board communications. Technical report, Volvo Technology Report (1998)
2. Ramamritham, K.: Real-time databases. International Journal of distributed and Parallel Databases (1993) 199–226
3. Kuo, T.W., Wei, C.H., Lam, K.Y.: Real-Time Data Access Control on B-Tree Index Structures. In: Proceedings of the 15th International Conference on Data Engineering. (1999)
4. Lu, H., Ng, Y., Tian, Z.: T-tree or b-tree: Main memory database index structure revisited. 11th Australasian Database Conference (2000)
5. Litwin, W.: Linear hashing: A new tool for file and table addressing. In: Proceedings of the 6th International Conference on Very Large Databases. (1980)
6. Astrahan, M.M., et al.: System R: Relational Approach to Database Management. ACM Transactions on Database Systems **1** (1976) 97–137
7. Nyström, D., Tešanović, A., Norström, C., Hansson, J., Bånkestad, N.E.: Data Management Issues in Vehicle Control Systems: a Case Study. In: Proceedings of the 14th Euromicro Conference on Real-Time Systems. (2002)
8. de Riet, R.P.V., et al.: High-Level Programming Features for Improving the Efficiency of a Relational Database System. ACM Transactions on Database Systems **6** (1981) 464–485
9. Software AG / SAG Systemhaus GmbH: Adabas Database . (http://www.softwareag.com)
10. Birdstep Technology ASA. (http://www.birdstep.com)
11. Sleepycat Software Inc. (http://www.sleepycat.com)
12. Pervasive Software Inc. (http://www.pervasive.com)
13. Date, C.J.: An Introduction to Database Systems. Addison-Wesley (2000)
14. Lehman, T.J., Carey, M.J.: A Study of Index Structures for Main Memory Database Management Systems. In: Proceedings of the 12th Conference on Very Large Databases, Morgan Kaufmann pubs. (Los Altos CA), Kyoto. (1986)
15. Tešanović, A., Nyström, D., Hansson, J., Norström, C.: Towards Aspectual Component-Based Development of Real-Time Systems. In: Proceedings of the 9th International Conference on Real-Time and Embedded Computing Systems and Applications. (2003)