Active Database Implementation For Real-Time Computing

Mohamad Ridha
ridha72@yahoo.com

ABSTRACT

Real-time computing can be found in many different types of applications such as traffic control systems, telecommunication systems, manufacturing applications, stock market and trading transactions, military command and control systems. A real-time application that monitors its surroundings and is required to react immediately based on an occurred event need an active database system that is able to detect the event and condition to perform an action based on its certain rules specification. Compared to other systems, a real-time system requires not only correctness in its output and behavior, but also a prompt response in a certain time to avoid failure. Data received or stored by the system must be accessed and processed without risking any time constraint in real-time transactions. Both of these requirement and constraint become the main concern in implementing a database management system used in a real-time application. This concern has been a focus of research for the last decade in the field of databases and real-time computing. By studying previous research in this field, this paper explores on basic concepts, issues and proposed solutions on implementing active database systems in real-time applications.

1. Introduction

Real-time computing can be found in many different types of application such as traffic control systems, telecommunication systems, manufacturing applications, stock market programs, and military control systems. A real-time system, in contrast to other systems, requires not only correctness in its output and behavior, but also a prompt response in a certain time to avoid failure. Data received or stored by the system must be accessed and processed without risking any time constraint in real-time transactions. When this real-time data is stored in a database, the application needs a database system to handle schemas, queries, transactions and concurrency protocols, and the storage management [7].

If real-time application is required to react immediately based on an occurred event, it needs an active behavior that is able to detect the event in a certain condition and trigger an action based on some rules specification. If this active behavior is implemented as part of the database system, the database system will then be categorized as being both active and real-time [10]. The problem arises when such active behavior is considered as a factor that may reduce the response time that is needed by a real-time application. The level of complexity in implementing such database system increases considering the implementation should take into account both active and timeliness requirement for the application. This concern has been a focus of research for the last decade in the field of active databases and real-time databases. While there have been some good numbers of research works that focus on active and real-time database systems in separate topics, only few that discuss the active real-time database system as an integrated subject [6].

This paper is written based on several research papers in both fields of active and real-time database systems, with the goal to explore concepts, issues and proposed solutions in implementing an active real-time database system. The next part of the paper will be divided into several sections: section 2 covers the basic concepts on active database system, section 3 covers the basic concepts of real-time databases, and section 4 discusses issues in combining features of active and real-time database systems, with some possible solutions to these issues. List of references is given at the end of the paper.

In this paper, the use of word database refers to the collection of related data, and the word database system refers to a collection of programs that maintain the database (process queries, data access, etc.) together with the database itself [9].

2. Active Database System

2.1 Definition

An active database system is “a database system that is capable of initiating actions” [6]. Conventional database system is considered passive in which it relies on the user or application program in executing queries [11]. User submits query and the database system executes the query and returns the result. However, this passive behavior may not be suitable for many applications where certain condition in the database has to be monitored in order to avoid unwanted consequences. For example, in an inventory control application, the number of items in a product table in the database needs to be reduced based on the number of order placed by customer in any given time. The update query to reduce this number is run by the application program when the order has been placed. Before the update, another query needs to run to check whether the item is still available, i.e. the item quantity is not equal to zero, to avoid shortage of items or placing order when there are no more items. If the number of items is equal to or greater than the number of order, the order can be placed and the

update query is then run. Otherwise, if the check result returns zero, the order cannot be placed and the program needs to report the situation or send a reminder for reordering the items. The problem with this approach is the application program becomes less modularized with additional functionality to check the items availability prior to every purchase. The application program will also become less reusable which then result in costly maintenance [9].

Another way to accomplish this is by monitoring the condition with the use of a program that periodically runs queries to check the condition, which is the availability of the purchased item in the above example. The database is polled frequently and based on a certain rules specification, condition is evaluated and action is taken when the condition evaluated is true. If for any given time, the item is equal to zero or falls in a range of predefined values, the program will call another program that generates report or sends reminder. Since the polling is performed based on a frequent basis, the optimal frequency is difficult to determine. If the polling is too frequent, the database must process queries which return often nothing. The database will then experience thrashing which results in unnecessary waste of resources [11]. If the polling is less frequent, some events and an important response time window can be missed [9].

In contrast to traditional database system, active database system, as its name implies, is designed to have active behavior in which it does not passively rely on the user or application program in executing actions such as queries or stored procedures. The database system monitors a predefined condition based on arrival of an event or set of events which then sets a trigger to perform action. In the above example of inventory control application, a rule can be constructed by specifying the event to detect (an update to the product table), the condition to evaluate (number of items in the product table is equal to a value or a set of values) and the action to perform (call the procedure to send report or reminder). By giving the ability to the database to detect event, to evaluate condition and to take action, rules and constraints can be defined centrally in the database system consistently for all application programs. This will prevent the situation where two application programs can have inconsistency rules or constraints from happening [12].

2.2 Event-Condition-Action

The active behavior in active database system is managed by ECA (Event-Condition-Action) rule, where upon the detection of event E and the satisfaction of condition C, the action A is executed. While the event specifies the need to check, the condition determines what to check [12]. The semantic of ECA rule can be written [3]:

on *<event E>*
if *<condition C>*
then *<action A>*

For an inventory control application example, a rule can be specified as [5]:
define rule *R1* ***on*** *Product table update*
if *Product.ItemQuantity <= MIN_AMOUNT*
then *SendReport()*

It was suggested that rules are implemented as objects so the database system can easily manage them (in creating, editing, or deleting them) by using transaction mechanisms like any other object in the database [12].

2.2.1 Event

An event in the active database context is defined as “something that happens at a point in time” [3], not over a period of time, and it is instantaneous and atomic [5]. An event can be categorized into two: primitive events and composite events. *Primitive* events are elementary events

predefined in the system, and can be classified as follows [12]:

- *Database events*: Events that correspond to database operation, such as begin and end transaction, and access to data (retrieval, insertion, update, deletion).
- *Temporal events*: Events that are triggered at predefined points in time. They can either be absolute (e.g. triggered every midnight) or relative temporal events (e.g. 10 minutes after some other event).
- *Explicit events*: Events that are external to the database, raised by the user or the application. For example, readings from a sensor in real-time application.

Composite events are a set of events which contains primitive or composite events combined with logical operators such as conjunction ("and"), disjunction ("or"), or sequence [5], [12]. Primitive and composite events are demonstrated in the example below [5]:
define *event E1* ***on after*** *Product::Update*
define *event E2* ***on after*** *Order::Update*
define *event E3* ***as*** *(E1 and E2)*
define *event E4* ***as*** *(E1 or E2)*

Events E1 and E2 are primitive events, and events E3 and E3 are composite events with different logical operators. Although composite events are useful indicators for detecting complex situations, they can be very costly in regards to the time taken to detect them [4]. This factor will be one of the considerations when implementing an active database system for real-time application.

2.2.2 Condition

Before any action can be executed, condition specified in ECA rule must be evaluated. If the condition is true then the specified action is carried out, otherwise no action is performed. Condition can be a query to the database that returns a result or empty, or a call to an existing function or procedure with some return value [12]. The evaluation of condition can be done right away after the event is detected or after the transaction which triggers the event has been completed. In the example of inventory control application above, the condition is item quantity is less or equal to predefined minimum value.

2.2.3 Action

Action is carried out only after the evaluation of condition is satisfied. Action may include the database operations such as retrieval, insertion, update, or deletion of data [12], or call to stored procedure or function to execute several transactions. In the example of inventory control application, the action is calling a function to send report or reminder that the item quantity has reached the minimum.

2.3 Execution model

2.3.1 Coupling modes

Coupling modes can be seen as types "to determine how rules to be executed relative to the triggering transaction" [6]. In ECA rules, the occurrence of event, caused by a transaction, always comes prior to condition evaluation and action execution. There are three coupling modes where both condition evaluation and action execution can take place relative to the triggering event [12]:

- *Immediate*: where they both take place upon the arrival of the event. The triggering transaction, i.e. the transaction that causes the event occurs will be suspended until condition is evaluated and the action is performed or not depending upon the result of condition evaluation.
- *Deferred*: They both take place at the end of the triggering transaction before it commits.

- *Detached*: They both are performed in one or separate transaction.

2.3.2 Cascade triggering

Cascade triggering is a situation when an event triggers an action which in turn triggers another event that triggers another action. This situation can result in an infinite circular triggering. One way to avoid this to happen is to set a limit in triggering depth, or to prohibit any rule that may introduce such cascade triggering [12].

2.3.3 Conflict resolution and priorities

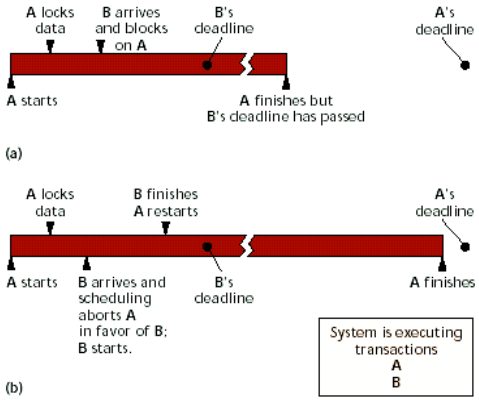
Conflict between rules can happen when an event is specified in several rules in the database system. In this situation, active database system needs to identify which rule should run first from the other. One way to solve the conflict is by using some form of priorities to either pick one rule or set the order of several rules. These priorities can be set by user or set by the database system by default, i.e. based on the creation time: the older, the higher priority [12].

3. Real-Time System

3.1 Definition

Real-time system is defined as "a system where not only the logical result of executing a task is important, but also there is a strict requirement imposed on the system to produce the result in timely fashion [5]". Real-time system does not mean fast system. Fast computing targets at minimizing the average transaction's response time, while real-time aims to meet the timing constraint requirement beside valid transaction execution [7]. Data received or stored by the system must be accessed and processed without risking any time constraint in real-time transactions. When this real-time data is stored in a database, the application needs a database system to handle schemas, queries, transactions and concurrency protocols, and the storage management [7].

Compared to conventional database systems, real-time database system expects the result of task execution to be delivered within a time constraint, called deadline. All or at least some transactions in real-time database system will have deadlines [1]. The database system can achieve this by making the task execution time predictable. One way to do this is by utilizing time-cognizant protocol in contrast to the 2PL (two-phase locking) protocol used in conventional database. In conventional database, as shown in Figure 1 (a), transaction A comes first before transaction B and locks data. B arrives later and waits for A to release its lock. A has not released the lock until after B's deadline has passed, which means B's deadline is missed. In real-time database, shown in Figure 1 (b), although B arrives later than A, B can start by aborting A since B's deadline is earlier than A's. When B finishes, A restarts and completes its task before its deadline [7].



(Figure 1)

3.2 Deadlines and temporal validity

Depending upon its varying degree of importance of meeting deadlines, real-time system can be categorized into hard

and soft systems [1]. In hard real-time system, no deadline should be violated and missing a deadline can cause negative or catastrophic consequences to the system or its environment. Soft real-time system which also aims to ensure tasks to meet their deadlines will not have a negative consequence if any deadline is missed. Real-time database system uses value function to give positive value if deadline is met and negative value if it is not met [2].

Real-time database system works in real-time data which are changing often. Much of the data become valueless after a brief period of time and cannot be used for obtaining new data or decision making process. This is called "temporal consistency requirement" in real-time database system which can used to determine a frequency of readings (data capture) and can be categorized into two [12]:

- *Absolute validity*: Validity of data lies only between absolute points in time.
- *Relative validity*: Different data must be temporally consistent with each other for them to be valid.

3.3 Transaction and scheduling

In conventional database system, transaction manager should support transaction ACID (Atomicity, Consistency, Isolation, and Durability) property while maintaining the high throughput by executing transactions concurrently [2]. In most real-time database systems, in addition to ACID property, transaction manager should also support real-time timing constraint. This means, transaction manager should be able to reduce delays caused by blocking or logging done in pessimistic concurrency control protocols. Two-phase locking (2PL) protocol that is often used in conventional database system may not be suitable for real-time database system because it may introduce deadlock when some transactions are involved in a circular wait [2]. The use of locking found in pessimistic concurrency control should therefore be avoided; instead the use of optimistic concurrency control (OCC) is preferable since it is well-suited for real-time database system [2], [6].

In real-time database system, schedules are managed by a scheduler. Schedule can be predefined off-line (static) by user, or defined on-line dynamically by the scheduler at run time for given transactions. In the dynamic scheduling, when too many transactions are received at any given time, the scheduler might experience overload in determining their schedules within a certain deadline. To resolve this, some transactions can be aborted, or they can be blocked from coming into the system [12].

3.4 Contingency plan

As mentioned above, hard real-time system is designed to guarantee all deadlines are met. When there is a risk of a transaction missing a deadline, a contingency plan is defined to be executed when a deadline cannot be met [1]. The result of transaction defined in contingency plan might not be exactly the same as the original transaction, but it is completed before the deadline as required. This can be done by several ways [12]:

- by dividing the transaction into parts that can be skipped to decrease execution time,
- by dividing the transaction into parts that are mandatory and optional to be executed,
- by creating primary and alternative versions of the transaction.

There are overhead associated with the above that should be taken into consideration when choosing contingency plan techniques [12].

3.5 Predictability of timing

Real-time database systems requires timing predictability when executing a task or transaction. Several factors make it difficult to predict a deadline for a transaction in the real-time database system [12]:

- The data of which the transaction depend upon can change over time from execution to execution which makes it is hard to predict how long the transaction can complete its task. This can be handled by avoiding the use of some program constructs such as recursive or dynamic creation of data.
- The use of pessimistic concurrency control protocol can introduce locking and blocking for an unpredictable amount of time. Optimistic concurrency control protocol can be used instead .
- Reading data from disk is much slower than reading it from memory. This magnitude difference makes it hard to predict the timing of completion since a transaction does not know whether the data is still in the disk or already exists in the memory.
- Transactions can be aborted and restarted in the middle of on-going executions. This unpredicted number of aborts and restarts make it hard to predict the completion time.

3.6 Buffer Management

Buffer management is associated with the use of main memory during reading and writing data from and to the disk [8]. In a real-time system that has limited main memory, buffer management has to ensure that high priority transactions are executed [2]. When the main memory has a plenty of space, the database is preferred to reside in the main memory ("memory resident database system") to enable fast and predictable access [1], [2].

If the database resides in main memory, the problem arises when the system fails and causes the data stored in main memory lost. To handle this situation, a non-volatile or persistent storage is needed to keep the persistence data, or the memory can be made stable supported by power from battery. Logging needs to be performed as a way to do recovery in case the system fails [12].

4. Solutions for issues found in implementation

In section 2 and 3, some basic concepts of active and real-time database system are outlined. In this section, some issues in implementing active database system in real-time computing will be discussed.

The implementation can be viewed as merging the two database systems (active and real-time) into a one integrated, active real-time database system. The main problem is the existence of conflicting requirement of the original systems. Active database system requires an active behavior, and real-time database systems requires timelines and predictability. Active behavior adds execution time and adds some unpredictability factors to the system. To solve this problem, time constraint handling needs to be integrated with active requirement. To do this, time-cognizant priority assignment, concurrency control and conflict resolution mechanism need to be identified [13]. Some solutions are listed as follows:

4.1 Avoiding transaction time slack

To avoid the situation where the triggering transaction missing deadlines because of the time taken by triggered rules execution decreases its time to complete the task (and meet the deadline), the rules should be run in detached mode,

concurrently with the triggering transaction. [12]

4.2 Avoiding cascade triggering

Cascade triggering in real-time constraints environment can cause many transactions miss their deadlines. To avoid this, the database system should not allow rules to trigger another rule. [5], [12]

4.3 Avoiding high cost in composite event detection

As mentioned before, although composite events are useful indicators for detecting complex situations, they can be very costly in regards to the time taken to detect them. This is unacceptable in real-time system. To avoid this, composite events should only triggers rules with detached mode, which will not result in the unpredictable execution time of the rules. [5], [12]

4.4 Avoiding missing deadline during overload

As mentioned before, when scheduler experiences overload, some transactions must be aborted, hence they will not meet the deadlines. In hard real-time system, this should not be happening. To avoid this, contingency plan can be defined and executed when there is a chance that a transaction cannot meet the deadline and there is enough time for the alternate actions to take place. The time of switch can be identified by monitoring milestones set for transactions; when a transaction misses milestone, the contingency plan is carried out. [12]

4.5 Semantic for time-constraint ECA rule

The issue of time constraint is not considered the most important factor in active database system as in real-time application [5]. ECA rule specification does not consider this real-time constraint. To include the time constraint factor in ECA rule, the following semantic is proposed [5]:

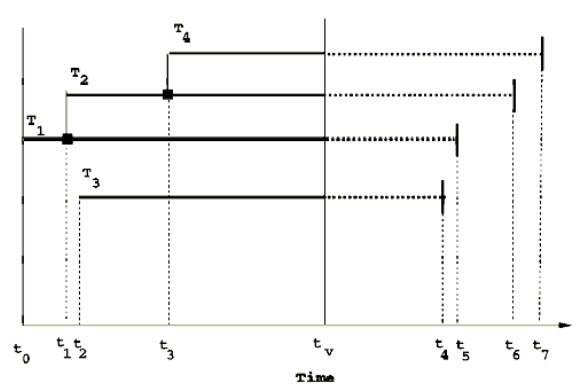
ON *<event E>*
IF *<condition C>*
DO *<COMPLETE> action A <WITHIN t seconds>*

The above semantics mean upon arrival of event E, condition C is evaluated and if it is true then the action A is executed and **should be completed** within t seconds. The semantics above should be understood differently from the semantics used in active database system:

ON <event E>
IF <condition C>
DO action A <WITHIN t seconds>
where action A is executed, not completed in t seconds.

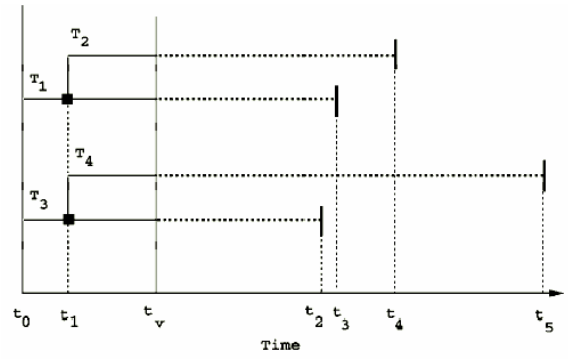
4.6 Optimizing the optimistic concurrency control

As mentioned earlier, locking problem can be caused by using the pessimistic concurrency control. The use of locking found in pessimistic concurrency control should therefore be avoided; instead the use of optimistic concurrency control (OCC) is preferable since it is well-suited for real-time database system [2], [6]. However, two drawbacks in using conventional OCC have been identified when used in active real-time database system [6]:



(Figure 2)

- In Figure 2 above, T1 starts at t0 with deadline t5 and triggers T2 at t1. T2 starts at t1 with deadline t6 and triggers T4 in t3. T3 starts at t2 with deadline t4. Since deadline of T1 > deadline of T3, priority of T3 > priority of T1 (based on the rule of "earliest deadline-first"). Assume T1 is in conflict with T3. In tv, when T3 enters validation phase, T1 will need to be restarted based on the rule of conventional Optimistic Concurrency Control protocol. When T1 is started, T2 and T4 will need to be aborted (since they are chain-triggered from T1). The restarted T1 will not have a chance to complete its deadline. By restarting T1, T2 and T4 are wasted. This **does not satisfy** the important requirement of real-time system in regards to timing constrint or deadline, where no transactions miss their deadline (in hard real-time system), and only really few transactions may miss the deadline (in soft real-time system).



(Figure 3)

- In Figure 3 above, T1 starts at t0 with deadline t3 and triggers T2 at t1. T2 starts at t1 with deadline t4. T3 starts at t0, at the same time with T1, with deadline t2. Since deadline of T1 > deadline of T3, priority of T3 > priority of T1. Assume T1 is in conflict with T3. When T3 enters validation phase at tv, T1 will need to be restarted based on the rule of conventional Optimistic Concurrency Control protocol. When T1 is started, T2 will need to be aborted (since it is triggered from T1). There are two alternatives of resolving the conflict:
- T1 is restarted as stated above. T2 will then be re-triggered and will have hard time completed its deadline in t4.
- T3 is restarted. T4 is re-triggered, and since its deadline t5 still is still long way, T4 has better chance to complete compared to T2.

The example above shows that in spite of the decision taken by the conventional OCC to chose the first alternative, there are two alternatives that can be considered, first when T1 is restarted and second when T3 is restarted, and second alternative is better in which the restarted triggered transaction (T4) completes its deadline.

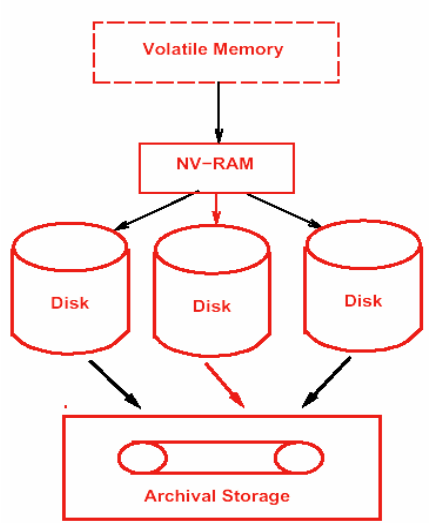
A new optimistic concurrency control (OCC) protocol, called OCC-APFO (Optimistic Concurrency Control-Adaptive Priority Fan Out) is proposed [6]:

"*OCC-APFO is an optimistic, priority cognizant protocal based on the dynamic adjustment of serialization order. OCC-APFO uses the notion of timestamp intervals to record and represent the serialization orders induced by concurrency dynamics. Timestamps are associated with both transactions and data items, but in different ways:*

- ***Data items and Timestamps**: Each data item has a read and a write timestamp, in addition to their usual meanings, i.e. the read and the write timestamps are the largest timestamp of transactions that have read or written the data item respectively.*
- ***Transactions and Timestamps**: OCC-APFO associates with each active transaction a timestamp interval expressed as [lower bound (lb), upper bound (ub)] pair. The timestamp interval denotes the validity interval of a transaction. The timestamp intervals are also used to denote serialization order between transactions."*

4.7 Using Non-Volatile Memory

As stated earlier in other section, the primary reason to place the data in main memory is to avoid unpredictable seek and rotational delays introduced by disks [7]. However, maintaining large amount of data in main memory can be very costly and a solid recovery mechanism that can handle a system failure should be in place considering all data will be lost because main memory is volatile. To reduce the time taken to read and write to persistence disk storage, especially in active real-time database system where the active behavior may triggers many transactions that contains read and write operations while the timing constraint is also enforced by the real-time requirement, non-volatile memory is suggested to be used to make deadlines can be met [13]. Non-volatile RAM (NV-RAM) can be used as disk cache where data can be moved from the main memory to it and from it to the disk. It can also be used as temporary storage for data to increase performance. Its data can then either be migrated to the disk or not depending upon the data characteristics which will explain next.



(Figure 4)

Figure 4 shows the four-level memory hierarchy proposed in [13]. Where the data is placed depends upon the characteristics of the data which can be categorized as follows [13]:

1. Temporal, and non-temporal data
2. Accessed frequently, and accessed rarely
3. Persistence, and non-persistence
4. Critical and non-critical

Temporal, accessed frequently, non-persistence and non-critical data can be placed in main memory, while non-temporal, accessed rarely, persistence and critical data should be placed either temporarily in NV-RAM or in the disk.

References

1. J. A. Stankovic, "Distributed Real-Time Computing: The Next Generation", *Journal of the Society of Instrument and Control Engineers of Japan,* January 3, 1992
2. B. Kao and H. Garcia-Molina, "An Overview of Real-Time Database Systems", *Advances In Real-Time Systems*, p. 463-486, 1995
3. N. W. Patton and O. Diaz, "Active Database Systems", *ACM Computing Survey,* Vol. 31, No.1, March 1999
4. J. Hansson and M. Berndtsson, "Active Real-Time Database Systems", *Active Rules in Database Systems*, p. 405-426, 1999.
5. M. Berndtsson and J. Hansson "Issues in Active Real-Time Databases", *Proceedings of the International Workshop on Active and Real-Time Database Systems*, 1995
6. A. Datta and S. H. Son, "A Study of Concurrency Control in Real-Time, Active Database Systems", IEEE Transactions on Knowledge and Data Engineering, Vol. 14, No. 3, May/June 2002.
7. J. A. Stankovic, S. H. Son, and J. Hansson, "Misconceptions About Real-Time Databases", *IEEE Computer*, 32(6):29-36, June 1999
8. R. Ramakrishnan and J. Gehrke, "Database Management Systems", 3rd Edition, 2003
9. R. Elmasri and S. B. Navathe, "Fundamentals of Database Systems", 2nd Edition, 1994
10. A. Bestavros, "Advances in Real-Time Database Systems Research", ACM SIGMOD, Vol. 25, Issue 1, p. 3-7, March 1996.
11. U. Dayal, B. Blaustein, A. Buchmann, I. Chakravarthy, et.al, "The HiPAC Project: Combining Active Databases and Timing Constraints", ACM SIGMOD, Special Issue on Real-Time Database Systems, Vol. 17, Issue 1, p. 51-70, March 1988.
12. J. Erikson, "Real-Time and Active Databases: A Survey", ARTDB-97, The 2nd International Workshop on Active, Real-Time, and Temporal Database Systems, Advance Proceedings, pages 195--216. IEEE, 1997
13. K. Ramamritham, R. Sivasankaran, J. A. Stankovic, D. T. Mosley, and M. Xiong, "Integrating Temporal, Real-Time, and Active Databases", ACM SIGMOD, Vol. 25, No.1, p.8-12, March 1996

Biografi Penulis

Mohamad Ridha

Lulus SMA tahun 1990 dari SMA Negeri 14 Jakarta. Sempat kuliah di Universitas Indonesia, Teknik Elektro, sebelum mendapatkan beasiswa sekolah ke Amerika tahun 1991. Menyelesaikan S1 tahun 1995 di Purdue University dan S2 tahun 2000 di Arizona State University pada jurusan Ilmu Komputer. Saat ini sedang melanjutkan studi dalam jenjang S3 pada jurusan yang sama. Berpengalaman lebih dari 9 tahun di bidang ilmu komputer khususnya rekayasa perangkat lunak (*software engineering*) dalam dunia akademi, riset dan industri di Indonesia dan Amerika.

Informasi lebih lanjut bisa hubungi:

ridha72@yahoo.com