The RAPID Case Study: Requirements for and the Design of a Fast-Response Database System

Antoni Wolski
Janne Karvonen
Anton Puolakka†

Technical Research Centre of Finland (VTT)
VTT Information Technology
P.O. Box 1201, 02044 VTT, Finland

`e-mail: <first name>.<last name>@vtt.fi`
`WWW: http://www.vtt.fi/tte/projects/rapid`

Abstract

Time-critical industrial applications represent a database requirement profile which is significantly different from that in traditional (business) database applications. Results of a corresponding needs survey performed among leading developers of factory systems are presented in the paper. The requirements have paved way to the development of the RAPID system–a fast-response and active history database Client/Server system which has a unique assortment of temporal and active characteristics. Some of the most crucial design questions are discussed, including the choice of the process architecture and concurrency models, the temporal table model and the triggers semantics.

1 Introduction

In a complex industrial installation (such as a paper mill, a power plant or a chemical process) a lot of data is produced by sensors and the automatic control system. In order to post-process the data, the system has to cope with the following data characteristics:

- the data assimilation rate may be as high as thousands measurements per second,
- data sets are modeled as *histories* [Jen+94],
- data changes require actions to be taken; an active behavior highly preferable,

In [Dat94], systems of the above type are called ARCSs (Active Rapidly-Changing-Data Systems), and the corresponding database components—ARCS databases. An ARCS database incorporates concepts of temporal, active and real-time databases.

In the RAPID project (1992–1995) we pursued adequate solutions for industrial ARCS databases. The work resulted in the construction of the RAPID Data Store System—a unorthodox Client/Server database system satisfying the most significant needs of certain industrial ARCS environments. In this paper, we summarize the needs surveys we performed in 1992 and 1993 (Section 2). In subsequent sections we discuss various design alternatives. Section 3 is devoted to issues of process architecture. In Section 4, we discuss multithreading and scheduling. Efficient implementation of temporal tables is dealt with in Section 5. New approaches to database triggers are presented in Section 6. In Section 7, we deal with selected issues related to the main-memory orientation in the database system design. A plan of future work concludes the paper.

2 Needs survey

2.1 Focus

After the RAPID project had been started at VTT Information Technology in 1992, almost forty companies were surveyed with respect to special needs for database technology in the development of industrial data processing systems. Some of the companies later joined the project steering group and vastly contributed to the eventual shape of the RAPID software.

The first step in the needs survey was to identify a prospective application area. An interesting observation was that, generally, the industrial partners were unwilling to consider using a real-time database to maintain the data

†Currently with Martis Oy, Finland.

Proc. First Wokshop on Real-Time Databases (RTDB'96),
March 7-8, 1996, Newport Beach, CA, U.S.A.

within an automatic control system (i.e. during the "automation phase" of the data life cycle). The reason was concern of predictability. Industrial control systems are, by nature, hard real-time systems. The partners strongly believed that hard real-time tasks should be scheduled statically (by pre-assigning each hard task a time slot), and the current specialized technology in the form of PLC (Programmable Logic Controller) equipment was best suited for the purpose. A database system with a dynamic transaction scheduler was considered too big a risk.

On the other hand, the partners brought our attention to one neglected field which was utilizing the process data during the *monitoring phase* immediately following the automation phase. During this phase, the data is collected, organized, post-processed and used in time-critical applications which are, however, not hard real-time in nature. The primary application area identified were *control room applications* in complex industrial installations, where the data is displayed for use by the operators and also automatically acted upon (e.g. to alert the operators), with the purpose of providing the human feed-back to the automatic control system.

The systems of the monitoring phase need not support deadlines. The rationale is that human beings are in the interaction loop of the system, and the human reaction time is unpredictable. There are, of course, performance requirements: the throughput and the response time on specific tasks. The timeliness of important operations is to be supported by prioritizing tasks on a "best effort" basis.

The RAPID Data Store System was designed to provide the database functionality to the monitoring phase applications. The following subsections contain the summary of the needs survey [JP93] and the subsequent requirement analysis [WJ93].

2.2 Transactions and functionality

The identified transaction types are listed below. The concept of a *command* is introduced to denote a database operation (possibly set-oriented) of a high-level database language like SQL.

SRT *State reporting transactions* [Dat94] which insert new sensor data into the database. The transactions perform "blind-writes" (they do not read the database) and are also "single writers" [Gra93] meaning there are no conflicting write operations. The unit of atomicity and isolation [GR92] is a command. The data written by SRTs is called *primary data*. All the other data is *secondary data.*

DRT *Data refining transactions* which read one data set and write another. Here, again, the transactions are "single writers" and there are, essentially no read-write conflicts with the SRT transactions because the two types of transactions deal with data items at different times (or, put it another way, use different versions). There may be read-write conflicts within the DRT class, though.

QT *Query transactions* which retrieve data for the purpose of display and reporting. Potentially, QT transactions engage in read-write conflicts with the DRT transactions. However, while the *snapshot consistency* may be sometimes useful, the *temporal consistency* [Ram93] of the data if more important (i.e. temporal validity prevails over serializability).

The above characteristics leads to disposing of traditional transaction management almost entirely. It was shown in [Gra93] that, in the presence of single writers, and the *loop-free data flow*, only simple mutual exclusion is needed, to deal with read-write conflicts, in order to achieve serializability. Additionally, in our case, a command-level rollback (atomicity unit is a command) is satisfactory to meet the atomicity requirements. One surprising survey result also is that no transaction-level durability is required—a checkpoint-based recovery is deemed sufficient.

The query patterns stress usage of time—either transaction time or valid time [Jen+94]. The data is fetched using time intervals, series of time points or simply the latest data is fetched. There is a need to join the data using valid time.

There is a broad requirement for active characteristics of a database. They may be articulated as a need for ECA rules [MCD89]. In addition to rules fired by elementary events, there is a need to support composite events, to be able to express, for example, delays and event repetitions.

2.3 Performance requirements

The performance requirements are summarized in the table below, in terms of data flows external to the RAPID DAS. In order to reflect the required performance/cost ratio, the requirements are set for a *reference equipment* corresponding to a typical Intel 486 PC (66 MHz) with a sufficient amount of memory. In a real system, the performance levels may be multiplied by scaling up the equipment.

The Data Store is expected to hold primary and secondary data half in half. Total number of sensor data items (not including timestamps and quality indicators) is expected to be up to 100 000 rows, in the reference equipment. It should be possible to archive half of this data into a disk-based database for further processing.

Data flow	**Characteristics**	**Performance** (q = query command) (v = value)
SRT transactions: inserting primary data	Inserting binary or/and analog values, high priority	10 - 500 v/s; 50/50 binary and analog
DRT transactions: inserting secondary data	Reading primary data and inserting secondary data (mostly analog), low priority	1- 10 v/s (analog)
QT transactions: retrieval requests	Responses to query commands: scalars, vectors, and time series, normal priority	1-10 q/s, 10 - 1000 v/s (up to1000 rows) resp. time < 0,5 s
Archived data	Portions of secondary and primary data, low priority	10 - 500 v/s
Data Definition Commands	Commands defining or modifying the data structures, very low priority.	<< 1/s

Table1. Performance requirements of different data flows.

2.4 Environment and interface requirements

The system is required to run on general-purpose computing platforms utilizing standard equipment (micro- and minicomputers) and common operating system like UNIX, Windows NT or OS/2. The industrial partners stressed also the issue of learning curve of a new technology. The data model should be simple, and the programming interface should be based on existing industry standards.

3 RAPID Design Summary

3.1 Process architecture

The performance requirements resulted in a main-memory based design. The question was whether the main-memory database would be encapsulated within a private address space of a single operating-system-level process, or allocated to memory shared by application processes or, perhaps, shared by specialized server processes. We have chosen the first alternative resulting in a database server process communicating with application processes using inter-process and network communications means. What prevailed was the reliability factor—the other alternatives would increase the probability of corrupting the database in connection with application failures. Another advantage is the versatility of the Client/Server architecture. However, the solution brings the penalty of the communications overhead, when compared with a shared-memory based solutions. The overall architecture of a RAPID application system is shown in Fig.1.

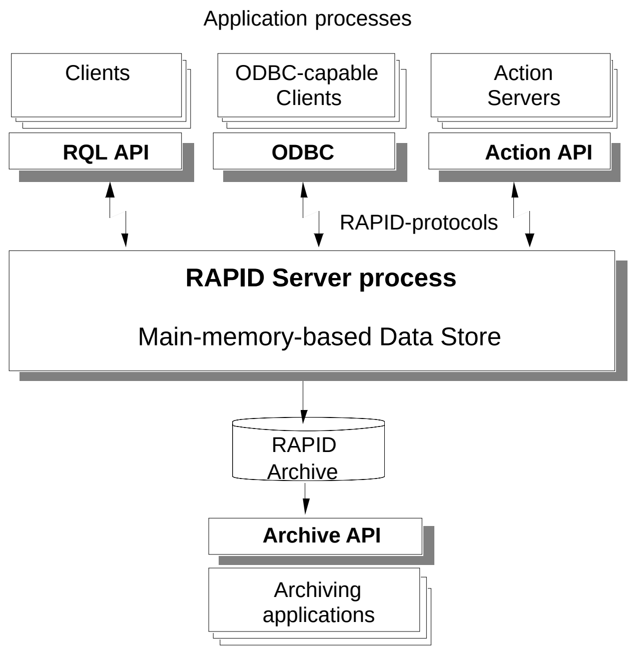

Fig. 1: RAPID system architecture–the server process and application processes.

The RAPID Client/Server protocols are implemented using the TCP/IP socket interface.

The RAPID Archive facilitates a long-term storage of RAPID data. It is a persistent collection of replicated histories implemented in disk files. The objects in the RAPID Archive have the semantics of a stream: a *history archive stream* is fed by the RAPID Server at one end and it is emptied by an archiving application (which is, for example, writing into an SQL database) at the other end.

3.2 Application interfaces

The RAPID application interfaces follow the model of function-based interfaces to SQL databases in that there is a platform-independent database language and a programming language binding in a form of a library. The RAPID Query Language (RQL) is a subset of SQL-92 [MS93], with temporal and active extensions. The programmatic interfaces are in the form of C++ class libraries.

RQL API is a library for programming RQL clients using the RAPID Server in a usual Client/Server fashion. It is modeled after the forthcoming standard ISO-ANSI SQL CLI (Call Level Interface) [CLI95].

Similar functionality is offered in the form of the *ODBC* driver for Windows-based ODBC-compliant applications.

In order to be able to receive action requests generated by RAPID triggers, the *Action API* library is used in an application.

The *Archive API* library is used to extract data from archive streams.

3.3 Concurrency

Typically, database servers have multithreaded design dictated by the need to maintain several active transactions in the same time and to handle I/O requests efficiently. In order to satisfy the presented requirements, the method of securing atomic writes and reads, in case of read-write conflicts, proposed in [Gra93], would suffice.

However, we have chosen an even simpler approach: the data manipulation requests are processed serially in a single main thread. In place of dynamic scheduling, a priority-based scheduler selects each next request to be processed. The scheduler uses an aging-based priority promotion scheme to guarantee that the service availability (response time) in CPU congestion situations is proportional to the assigned priority. Using of a single thread may, theoretically, lead to the priority inversion problem [Son95], but the severity of the phenomenon is diminished by the fact that, in reality, any request is executed in just few milliseconds (if the database size is not prohibitive).

There are also auxiliary threads in the server. They are used to perform I/O-based processing, like checkpointing, loading, dumping and automatic archiving of the Data Store into a disk-based database. The auxiliary threads introduce a concurrency control problem of its own. However, it is sufficient to maintain temporally consistent view of the database (i.e. a view of the history at a time point in the past) in auxiliary threads, which may be attained using the valid time information available for the histories.

All the threads run with equal priority, and the CPU scheduling is provided by the operating system of the computing platform.

3.4 History tables and temporal joins

The temporal requirements resulted in the design of *history tables*. For example, one creates a history table with the command:

```
CREATE HISTORY TABLE temp_meter (
     temp INT,
     quality CHAR(3) ) SIZE 10000
```

There is a built-in timestamp column called "OTS" (object timestamp), and each row in a history table is automatically timestamped with transaction time. The difference between a history table and a true temporal database table is that, in a history table, the timestamp column is accessible in the same way as any other column.

The difference between a history table and a relational database table is that the size (cardinality) of the history table is limited (to the default size or the size specified in the SIZE clause. Additionally, the rows in a history table have predefined order (chronological by transaction time). The table has also the nature of a circular buffer: new inserted data will always fit in; as a consequence the latest data will be removed from the table. In order to secure the data against such fate, the user must define objects called *RAPID archivers* which will replicate the history tables into persistent RAPID archive streams.

The valid time of data is calculated at the time a query is executed. The valid time of a row is the time period starting (inclusive) with the transaction time of the row and ending (exclusive) with the next transaction time recorded for the object. Temporal joins are based on valid times. A regular *temporal join* [Jen+94] may be requested with the command:

```
SELECT col1, col2 FROM tab1 TIMEJOIN tab2
     RETURN 24
```

returning 24 latest rows of the join. The TIMEJOIN keyword reflects the dual nature of history tables. It tells the system that the tables are to be treated as temporal tables in this query. In the RETURN clause, advantage is taken of the fact that result sets are also histories. They are ordered reverse-chronologically by default. In the RETURN clause, a user may specify thus the length of the result history.

In addition to regular joins, special *timepoint series joins* are supported whereby the result table is temporally joined with a set of time events given in an RQL command. On can request a timepoint series join in the following way:

```
SELECT col1, col2 FROM tab1 TIMEJOIN tab2
     TIMEPOINT SERIES INTERVAL '10' MINUTE
     RETURN 24
```

returning only such rows of the previous result which have valid times containing the time points specified using an interval. There are also other ways to specify the time points: using an explicit list of timestamp values and by way of a history table containing the timepoints.

Because the data in history tables is clustered by transaction time, no indexes are needed for timestamp columns, and time-based queries are performed efficiently using binary search. Thanks to the same fact, it is possible to implement temporal joins efficiently: essentially, the sort-merge join algorithm is used, where the data sets are already pre-sorted.

3.5 RAPID triggers

RAPID triggers are ECA rules. The trigger syntax is based on the existing SQL3 proposal [SQL3]. A crucial question was how to accommodate composite events. It has been proposed [GJS92] to model composite events as finite state automata having a special "accepting" state. Upon entering this state, the composite event "fires", i.e. stimulates the condition evaluation and the subsequent action execution.

One problem with the above approach is that a specification of a general purpose automaton requires a complex notation which is not intuitive for an end-user. A "user-friendly" solution was chosen where composite events are typified, and there is a specification syntax for each supported composite event type automaton. This results in a simple and intuitive notation.

A minor departure from the model of [GJS92] is that there is no "accepting" state in the automaton. Instead, some transitions to the idle (initial) state result in a "firing" of a trigger. Because, in the same time, the trigger assumes the initial state, no additional transition is necessary after the firing event.

The framework of the RAPID trigger definition command is the following:

```
trigger-definition::=
     CREATE TRIGGER trigger-name
     event-type
     ON table-name
     [event-specification]
     [WHEN (trigger-condition)]
     (action-list)
```

The syntax above follows the SQL3 scheme except for the even-specification clause. It meant to contain the state transition information when the event-type represents a composite event. Let us have a look at the RAPID event types:

```
event-type ::=
     {INSERT
     | {UPDATE [OF column-name ...]}
     | WRITE
     | TIMER
     | COUNTER}
```

The last two ones represent composite events: the first one related to delays and the second one related to recurring events.

For example, the following trigger is fired by an event of type TIMER, when a value inserted into a history table exceeds 100 and does not drop below 90 within 10 minutes. When fired, the trigger launches an action (an alarm procedure call) if the condition (the quality of the latest data is OK) is satisfied at that moment .

```
CREATE TRIGGER temp_alarm
TIMER
ON temp_meter
   SET INTERVAL '10' MINUTE
   START ON INSERT (value > 100)
   CLEAR ON WRITE  (value < 90)
WHEN (quality = 'OK')
(activate-overheating-alarm@action-handler)
```

The functioning of the timer automaton is illustrated in Fig.2. Two case are shown: when the timer fires by timeout (B) and when it is cleared (A). The example represent a very common surveillance activity in control room applications.

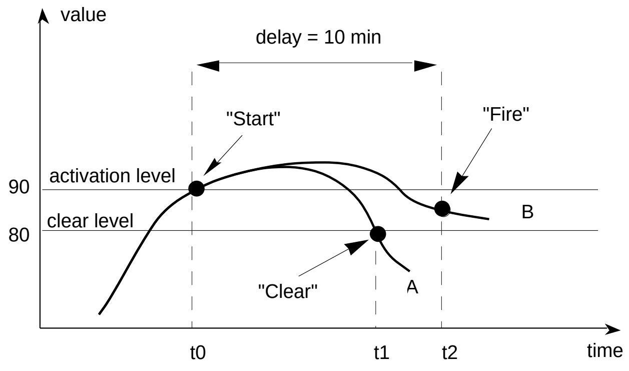

Fig. 2: Illustration of a timer trigger operation.

In RAPID, an action was decided to be an arbitrary external procedure called at a dedicated action handler process. In the action specification syntax:

```
action-spec::= action-name@action-server-name,
```

action-server-name is a name of an action server which the action server process supplies upon connecting to the RAPID Server, and action-name is a name of a procedure in that action server.

The decoupled action execution model was chosen because multi-command transactions are not supported. In other respects the RAPID triggers support the standard trigger mechanism semantics [WC95]

The following is an example of a counter-based trigger. It generates a notification when four consecutive values are recorded above a certain value level and the reset level was not reached (Fig. 3):

```
CREATE TRIGGER repeated_overflow
COUNTER
ON furnace
     SET 4
     INCREMENT ON INSERT (temp > 90)
     RESET ON INSERT (temp < 80)
(RepeatedUpperAlarm@TempActions)
```

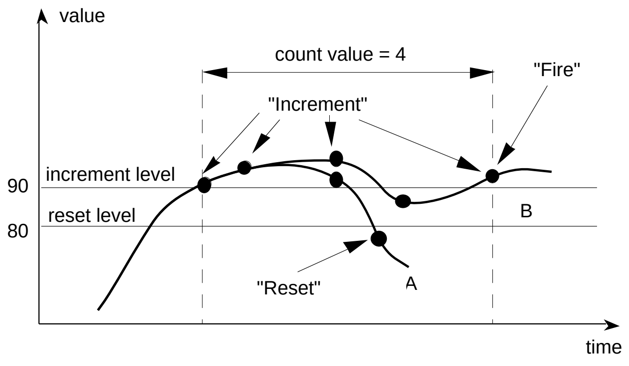

Fig. 3: Illustration of a counter trigger operation.

The full syntax of the RAPID triggers is included in the Appendix.

3.6 RAPID archivers

RAPID archivers are periodically active objects which are responsible for replicating selected history tables into the RAPID archive streams. The definition of an archiver includes the specification of a history table it is defined on, a mapping of names and column formats into a relational database table, an interval between the archiver activations, and a directory path to the location of the archive streams. The following is an example of an archiver definition:

```
CREATE ARCHIVER m23_archiver
     OF TABLE meter23 TO device23
     COLUMN timestamp TO tstmp CHAR
     COLUMN alarm_flag TO flag INT
     COLUMN meas_value
     INTERVAL'10'MINUTE
     USING PATH '../archives'
```

3.6 Ramifications of the main-memory based design

Contemporary computer architectures inflict difficulties in designing efficient memory-intensive programs. Especially, in a main-memory database, the traditional assumption that the cost of the memory access is constant [GMS92], does not apply any more. The reason is the hierarchical memory access architecture present in most modern processors, incorporating multi-level caches. Careful studies and experiments [Oks95] reveal great differences in program performance, depending on how well the *reference locality* is maintained in a program.

We addressed the problem of optimizing the locality of reference in various parts of the RAPID server software. The history table and the index structures are optimized for better locality of reference. Other algorithms are also scrutinized from this standpoint. The downfall of this approach is that automatic memory management facilities of programming systems (like dynamic object allocation on the heap in C++) have to be abandoned, and arcane programming techniques have to be introduced.

3.7 Performance

The current version 5.0 of RAPID runs under UNIX (HP-UX) and Windows NT operating systems. The performance tests revealed that the row insertion rate on a reference equipment (66MHz i486 PC) under Windows NT was 200 inserts per second while the sustained archiving activity was about 100 rows per second in the same time. The insertion load was fed through the network, one command and row per message. A user has various options to optimize the message traffic, e.g. by using multi-row INSERT commands or by buffering commands at a client site. The effects of these techniques has not been tested yet.

In UNIX environment (HP 9000) insertion rates above 1000 per second were achieved.

The effect the triggers have on the overall performance depends on how many trigger firings result in sending an action request. If triggers do not cause action requests, they have only minor effect on performance–it may be measured within few percent. However, each action request "costs" as much as three insertions.

4 Related work and conclusions

Various component technologies present in RAPID have been researched to a great extent. These include languages for temporal databases, like TSQL/2 [Sno+94], languages for composite event specifications like in Snoop [CM93] and Ode [GJS92], issues of temporal consistency [SL92, Ram93], main-memory based storage design [GMS92, JLRS94] and real-time scheduling [Son95].

The RAPID project has been an engineering laboratory of integrating these technologies into a workable solution meeting requirements of industrial users. The RAPID lesson is that, in order to meet primary requirements, it is sometimes necessary to sacrifice the overall generality. The RAPID Data Store System incorporates an active and temporal database, and also meets certain timeliness requirements, but each of the "personalities" has room for improvement in terms of generality.

Future work will concentrate on enhancing temporal capabilities towards a full-function temporal database and active functionality to enable a more general specification of composite events. A possibility to incorporate an object-oriented database language will be studied too. The challenge is to achieve the goals without compromising the overall performance. Also, an interesting engineering issue is how to produce database systems specialized for different applications easily. In order to achieve this goal, a component-based or extensible system approach has to be taken.

Epilog

At the time of this writing (February 1996), the RAPID software is being field-tested at three industrial partner companies in Finland.

References

[BHG87] P.A. Bernstein, V. Hadzilacos and N. Goodman. Concurrency control and recovery in database systems. Addison-Wesley Publ. Comp., 1987.

[CLI95] ISO-ANSI Working Draft, SQL Call-Level Interface (SQL/CLI), doc. no. ISO/IEC JTC 1 / SC 21 N 9464, March 1995. Previously as X/Open Specification: X/Open Document Number S203, Berkshire, United Kingdom, July 1992.

[CM93] S. Chakravarthy and D. Mishra. Snoop: An Expressive Event Specification Language For Active Databases. Tech. Report UF-CIS-TR-93-007, University of Florida, March 1993.

[Dat94] A. Datta. Research Issues in Databases for ARCS: Active Rapidly Changing Data Systems. *SIGMOD Record*, Vol. 23, No. 3 (September 1994), pp. 8–13.

[EC75] K.P. Eswaran and D.D. Chamberlain. Functional Specifications of a Subsystem for Data Base Integrity. *Proc. 1975 VLDB Conf.*, Frammigham, Mass., September 1975.

[GJS92] N.H. Gehani, H.V. Jagadish and O. Shmueli. Composite Event Specification in Active Databases: Model & Implementation. *Proc. VLDB'92 Conf.*, pp. 327-338.

[GMS92] H. Garcia-Molina and K. Salem. Main Memory Database Systems: An Overview. *IEEE Transactions on Knowledge and Data Engineering*, Vol. 4, No. 6, December 1992, pages 509 - 516.

[GR92] J. Gray and A. Reuter. Transaction Processing Systems, Concepts and Techniques. Morgan Kaufmann Publishers, 1992.

[Gra93] M.H. Graham. How to Get Serializability for Real-Time Transactions without having to pay for it. *Proc. Real-Time Systems Symposium*, Raleigh-Durham, North Carolina, December 1993, pp. 56-65.

[Jen+94] C.S. Jensen et al. A Consensus Glossary of Temporal Database Concept. *SIGMOD Record* Vol. 23, No. 1 (March 1994), pp. 52–64.

[JLRS94] H.V. Jagadish, D. Lieuwen, R. Rastogi, A. Silberschatz. Dalí: A High Performance Main Memory Storage Manager. *Proc. VLDB'94 Conf.*, Santiago, Chile, September 1994, pp. 48-59.

[JP93] J. Jokiniemi and A. Palomäki. Real-Time Databases: A Needs Survey. Research Report No. J-15, Lab. for Information Processing, VTT, Helsinki, February 1993. URL: ftp://ftp.vtt.fi/pub/projects/rapid/needs.ps.

[MCD89] Dennis R. McCarthy and Umeshar Dayal. The Architecture Of An Active Data Base Management System. *Proc. 1989 ACM SIGMOD Conf.* (Portland, Oregon, USA), pp. 215-224.

[MS93] J. Milton and A.R. Simon. Understanding the new SQL: A Complete Guide. Morgan Kaufmann Publishers, San Mateo, California, 1993.

[Oks95] K. Oksanen. All RAM Is Accessible, But Some RAM Is More Accessible Than the Other. Unpublished manuscript, URL: http://www.cs.hut.fi/~cessu/noram.ps.

[Ram93] K. Ramamritham. Real-Time Databases. *Distributed and Parallel Databases*, Vol. 1, No. 2(April 1993), pp. 199–226.

[SL92] X. Song and J.W.S. Liu. How well Can Data Temporal Consistency be Maintained. Proc. IEEE Symp. on Computer-Aided Control Systems Design, 1992.

[Sno+94] R. Snodgrass et al. TSQL2 Language Specification. *ACM SIGMOD Record*, Vol. 23, No. 1 (March 1994), pp. 65-86.

[Son95] S.H. Son (ed.). Advances in Real-Time Systems. Prentice-Hall, Inc., 1995.

[SQL3] Working Draft Database Language SQL3, J. Melton (ed.), August 1994, ANSI X3H2-94-329, ISO DBL:RIO-004.

[WJ93] A. Wolski and J. Jokiniemi. RAPID Data Store System: Requirements. Internal project report, Lab. for Information Processing, VTT, Helsinki, June 1993. URL: ftp://ftp.vtt.fi/pub/projects/rapid/require.ps.

[WC959] J. Widom and S. Ceri (eds.). Active Database Systems: Triggers and Rules For Advanced Database processing. Morgan Kaufmann Publishers, Inc., 1995.

Appendix: The full syntax of trigger definition in RAPID.

create-trigger-statement ::=
 CREATE [OR REPLACE] TRIGGER trigger-name
 event-type
 ON table-name
 [timer-specification | counter-specification]
 [WHEN (trigger-condition)]
 (action-list)
 [SEND ROW]

Trigger-name defines a name for the event-condition-action rule. Trigger names are unique within a datastore.

ON table-name defines the table into which the trigger is attached.

The WHEN clause contains the condition which is the condition to be evaluated before an action is activated.

event-type ::=
 INSERT
 | {UPDATE [OF column-name ...]}
 | WRITE
 | TIMER
 | COUNTER

INSERT defines that the trigger is fired on inserted values on the table called table-name, UPDATE defines that the trigger is fired on updated values, and WRITE defines that the trigger is fired on both inserted and updated values. TIMER defines a timer trigger whose behavior is defined in timer-specification. COUNTER defines a counter trigger whose behavior is defined in counter-specification.

OF column-name defines that the condition is checked only if the column-name column (or columns) is affected in the course of executing an UPDATE. If the column clause is not used, the trigger is fired whenever the table is updated.

timer-specification ::=
 SET interval-literal
 START ON INSERT {(trigger-condition) | ALWAYS}
 [CLEAR ON INSERT { (trigger-condition) | ALWAYS }]
 [CLEAR NOW]

interval-literal ::= INTERVAL 'number' SECOND | MINUTE | HOUR | DAY

Timer trigger is a normal trigger expanded with entity called timer. INTERVAL defines the time span that the timer will measure when activated. START ON INSERT defines when the timer is started. If trigger-condition is not specified, the timer will be started on every insert to specified table. Otherwise the timer is started only if trigger-condition is satisfied. After the timer is started, it is in active mode and it does not evaluate START ON INSERT condition any more. CLEAR ON INSERT defines when an *active* timer is stopped. It works the same way as START ON INSERT. If the timer is stopped while it is in active mode, the trigger returns to the idle state and reacts only to events defined in START ON INSERT. If the time span specified runs out and the trigger was not cleared, the action will be launched. If there is a WHEN condition specified, the trigger will not launch an action unless the condition is satisfied. CLEAR NOW is used in connection with the OR REPLACE option to clear the timer while redefining the trigger.

counter-specification ::=
 SET number
 INCREMENT ON INSERT (trigger-condition) | ALWAYS }
 [RESET ON INSERT (trigger-condition)]
 [RESET NOW]

SET defines the count value the counter fires at. The counter is incremented each time the condition in INCREMENT is satisfied. The counter is reset (set to zero) when the condition in RESET is satisfied. RESET NOW is used in connection with the OR REPLACE option to reset the counter while redefining the trigger.

trigger-condition ::=
 {comparison-condition [AND comparison-condition]}
 | between-condition
 | null-condition

comparison-condition ::=
 [OLD. | NEW.] search-column-name comparison-operator literal
 | literal comparison-operator [OLD. | NEW.] search-column-name

between-condition ::= [OLD. | NEW.] search-column-name
 BETWEEN literal AND literal

null-condition ::=
 [OLD. | NEW.] search-column-name IS NULL
 | [OLD. | NEW.] search-column-name IS NOT NULL

If the [OLD. | NEW.] is not used, the NEW column value is assumed. NEW denotes the row resulting from the triggering command. OLD denotes the image of the row before executing the UPDATE command or, in case of the INSERT command, the chronologically preceding row in the history table.

The actions are specified in the action list:

action-list ::= action-name@action-server-name [, action-name@action-server-name]…

The notation action-name@action-server-name defines the name of the action to be called and the name of the corresponding Action Server. Each Action Server submits its name when connecting the a RAPID Server. An action function has a standard parameter list including: the trigger name, table name and the key of the row the trigger is fired on (the triggering row).

The action-list is valid regardless of the existence of the corresponding Action Servers. If an Action Server is in the list and it is not connected at the time of the trigger execution, the action request is skipped and an error message is generated in the System Log.

A trigger is fired for each affected row. The data passed to action functions is thus row-specific.

SEND ROW causes the contents of the triggering row to be sent, following the action function call. The action function may extract the contents of the row using appropraite programming interface methods.