DESIGN AND IMPLEMENTATION OF A TEMPORAL TRIGGER SUBSYSTEM FOR THE TRIGGERMAN ASYNCHRONOUS RULE PROCESSOR

BY

NABEEL I. AL-FAYOUMI

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

1998

Dedicated To:
My dearest: *father* and *mother*,
My beloved *wife*,
And my precious *Ibrahim*

ACKNOWLEDGEMENT

I extend my thanks and deepest gratitude to my advisor Dr. Eric Hanson who spared no effort to help me through. I am especially grateful for the precious time he sacrificed and the valuable ideas he provided to lead me through the rough ride of the Ph.D. Also, many thanks to my Ph.D. committee members, Dr. Stanley Su, Dr. Herman Lam, Dr. Howard Beck, and Dr. Sanjay Ranka, who sacrificed plenty of their time and offered precious advice that helped me throughout my research. I cannot forget to thank our dear secretary Mrs. Sharon Grant who is always willing to help in any way she can.

I stretch out a million thanks, the deepest gratitude, and the greatest respect to my parents, my wife, my brother, and the rest of my family for their unconditional love and moral support. Their ceaseless love and support supplemented me with great strength and persistence that enabled me to overcome many obstacles and survive the dark moments I often passed through during my studies.

Special thanks go to my sponsors The Fulbright commission and Yarmouk University / Jordan, who provided substantial financial and logistic support during my stay in the states.

Finally, I would like to thank all the friends and colleagues who stood by my side, were there to help, and extended their hands whenever I needed it. I wish all of them bright futures and successful lives.

TABLE OF CONTENTS

LIST OF FIGURES

Figure

LIST OF TABLES

Table

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

DESIGN AND IMPLEMENTATION OF A TEMPORAL TRIGGER SUBSYSTEM FOR THE TRIGGERMAN ASYNCHRONOUS RULE PROCESSOR

By

Nabeel I. AL-Fayoumi

August,1998

Chairman: Eric N. Hanson
Major Department: Computer and Information Science and Engineering

This work presents a novel approach to developing a temporal active rule system as part of an asynchronous rule processor. Traditional database systems are passive with respect to data management. They are exclusively employed for data storage and manipulation. Recent technology advancements lead to more complex database applications. This compelled database systems to become more sophisticated in order to cope with the growing requirements of such applications. Advanced applications require database systems to be aware of the data being stored and take an active role by triggering actions when necessary. Hence, the idea of active databases has evolved and prospered.

The rapid advancement of information technology recently has inspired new database applications that are even more sophisticated. Specifically, many prevalent applications consider temporal attributes as a cardinal dimension of data. Such

applications opt for recognition and manipulation of temporal attributes, and crave for rules that would operate on them. There are numerous examples of applications that can make great use of active temporal systems such as retail marketing, credit card tracking and monitoring, medical research, stock market analysis, and so on. In addition, we conjecture that in the near future temporal attributes will be an inescapable and fundamental dimension of data.

Most temporal rule processing research projects have developed complex languages which unfortunately suffer from procedurality in their languages. This yields poor readability and maintainability. Moreover, it requires highly skilled users to utilize such systems.

The above considerations inspired us to design and develop a temporal rule system that avoids the shortcomings of other systems. The two main goals of our research are 1) to develop an efficient temporal rule system that would transcend the available systems in its temporal capabilities and 2) to carry through a high level of user-friendliness. Our work illuminates the language, an essential issue that others overlooked. We strove to furnish users with a language that is declarative and easy to read and administer. Other priorities include fast response, high performance, and versatility of use. In summary, this dissertation presents a framework for an efficient temporal rule processing system, and focuses on the major contributions of the prescribed system.

CHAPTER 1
INTRODUCTION

Rule processing systems have become components of next-generation database systems to permit them to fulfill the requirements of new applications. Temporal databases and rule processing systems are becoming attractive for applications such as stock brokerage, credit card account tracking, market-basket analysis, medical research studies, etc. Presently, most time-sensitive (temporal) applications are developed as external software modules that interact with the DBMS, but do not get much direct support from it.

Temporal database systems (TDBMSs) have been around since the mid-eighties. Since TDBMSs are well understood, commercial products of such systems have been developed and are currently in use. Active database systems (ADBMSs) have been studied throughout the past decade and have reached a relatively mature stage [Chak89, Ston90, Geh91, Geh92a, Wid92, Coup94, Hans96a]. Many commercial database systems have incorporated rule processing. Temporal rules–which hybridize temporal and active concepts–have become items of interest recently. Although temporal rule systems are not only interesting but also greatly needed, research in this field is still in its cradle stage.

Until recently, research projects have provided a complete framework for temporal active rule systems that are more idealistic than practical, and are far from being user-friendly. As we describe later, the focus of the research efforts in this area was to develop complete operational systems, even though their complete solution set is rarely

utilized in practice if ever. Consequently, the languages and the implementations are complex and their syntactical attributes are hard to understand and program. These qualities defeat the purpose of using such systems. Therefore, the research work introduced in this dissertation endeavors to mend the weaknesses of available active temporal systems and achieve fast response time, high efficiency, and most importantly, ease of use. A subsequent chapter illustrates by example the difference between TriggerMan's temporal language and other prevalent temporal rule languages. The examples clearly confirm the superiority of our language with respect to readability and ease of use.

The goal of a temporal active system is to allow quick and easy composition of temporal triggers to achieve an intended task. Accordingly, the model has to achieve completeness. In this work, the system provides only a subset of the complete solution set and can incrementally incorporate complementary subsets depending on the application. This is accomplished by providing extensibility tools that allow users (programmers) to add new types and functions. This approach reduces the anxiety about the system's functional completeness and leaves more space for concentrating on other more practically important issues.

This work defines a framework for an efficient, robust, and user-friendly temporal rule processing system. Attaining the power of a temporal rule system requires the maintenance of historical data. There are numerous ways and structures to represent historical data. Nevertheless, time-series representation has been the most popular model used to manage historical data. The two most popular representations for time-series are relational and object-relational. The relational model represents each data point as a tuple

with a field designated to hold its timestamp and one or more additional fields to hold associated data. The timestamp is the real-world time at which the value of the tuple becomes evident. TriggerMan adopts the more efficient object-relational model, which can represent a temporal attribute (a field of a tuple) as a time-series if appropriate. The time-series can be either built-in or a special extended object class (e.g., called a DataBlade in Informix [Info97], and a data cartridge in ORACLE [Orac97]).

The core of this dissertation is the conceptual design of the TriggerMan's temporal rule processing system. Throughout this dissertation, issues are discussed in detail according to their importance and relevance. One fundamental computer science philosophy this research solicits is that the interface is one crucial element of a software system. Hence, we have dedicated a substantial portion of our effort towards making TriggerMan's language (including the temporal sublanguage) as simple, versatile, and user-friendly as possible. From that standpoint, one principal contribution of this work is the introduction of a *declarative*, SQL-like language for temporal rule definition and manipulation rather than a procedural, unfamiliar language. Another major goal is efficient, robust, and reliable maintenance of historical data and triggering based on that historical data.

With respect to related work, the boundaries of the proposed project touch several database areas, some of which are relational active databases, object-oriented databases, object-relational (strongly related to the former two fields), and temporal databases. However, the literal field of this research work that of active temporal databases (ATDBMSs). The related work chapter reviews all the aforementioned topics in order of relevance to establish a solid and complete background for our work. The following

chapters address the different related issues in order of significance. The discussions clearly state all the assumptions, and omit minute implementation details when they are irrelevant.

CHAPTER 2
RELATED WORK

This chapter provides a brief discussion of the background related to this dissertation's topic. Furthermore, we survey the contributions of others in the area of active temporal databases. A subsequent section describes how the concepts of the related fields discussed here impact our model, design, and implementation. Since the main objective of this project is to develop an efficient temporal rule processor as part of the TriggerMan system, we dedicate a section on the TriggerMan project.

2.1 Active Databases

Active database systems are DBMSs equipped with capabilities to process rules or triggers defined on the data [Day88, Jae95, Hans95, Wid96a, Wid96b]. Recently, there has been a growing interest in this area due to the increased demand for such systems. Active databases are well understood theoretically, and the needs and anticipated functions of such systems are clearly manifested [ACT96]. However, implementations of active databases are being revised and enhanced to reach the ultimate goal of producing a complete, efficient, and fast rule processing system that is fit for all possible realistic applications.

The mainstream models of active database systems correspond to the relational and object-oriented database models. The current (and possibly the future) trend in database systems is a hybrid of the former two models called the *object-relational*

database model. Experts in the area speculate that this model will dominate the market in the near future because it combines agreeable features from both parent models [Ston97]. However, most research systems in the literature are based on either the relational or the object-oriented model, and hence, many of the active database systems reviewed in this section fall into one of the two former categories.

2.1.1 Relational Active Database Systems

Some of the relational active database systems are based on concepts borrowed from AI production systems. The languages are also similar to those of typical production rule systems such as OPS5 [Brow85]. The following is a brief overview of the most well known and potentially relevant relational active DBMS research projects.

The **Ariel** [Hans92, Hans96a] project adopts a rule processing model that is based on the OPS5 production rule system. Currently this project has an efficient rule testing mechanism called *Gator* networks [Hans96b]. A Gator network is an optimized generalization of the original *Rete* networks [Forg82] used in OPS5. One unique feature of such a network is the use of a special data structure called *virtual-•* memory node for holding the qualified tuples instead of the fully stored node. A virtual-• node holds only the selection predicates instead of the real data tuples (like a view). This allows substantial memory savings when the selectivity factor is low and the size of the relation is large.

Postgres [Ston87, Ston90] is a next generation extensible active DBMS that has been commercialized as the Illustra Database systems, which is now a part of Informix Dynamic Server [Info97]. Postgres has a tuple-level rule system where the main feature

is the introduction of “DataBlades.” A DataBlade is a utility that allows the extension of the data types and functions of the system. Furthermore, Postgres’s rule system supports forward and backward chaining functionalities.

Starburst [Wid91, Wid92, Wid96b] is an extensible relational database system that was extended with a rule system at the IBM Almaden Research Center. The system’s rule processor is based on arbitrary database state transitions, contrasting with other rule systems that rely on tuple or statement-level updates. This means that even if a state change may involve several inserts, deletes, and updates, the rule system considers only the net effect of all those events rather than considering each as an individual event. The Starburst rule system executes rules actions in a deferred mode (just before the transaction commits). Nevertheless, it provides a few commands to force processing a rule within a transaction, which is useful in cases where the execution of the rule is a precondition for transaction completion.

Datex [Bran93] is another project that is similar in concept to Ariel. Datex differs by having large OPS5 production programs run efficiently on large DBMSs. The system achieves high efficiency by utilizing sophisticated indexing schemes.

Related projects include **Paradiser** [Dewa92], **RPL** [Delc89], and others. These projects have potentially contributed to the area of active relational databases and produced frameworks for more advanced features that may be utilized in future database systems.

2.1.2 Object-Oriented Active Database Systems

It is harder to develop a coherent picture that exhibits how object-oriented database systems (OODBMSs) can be extended into active database systems. This is due to their recency and lack of standards. Therefore, the literature displays a wide spectrum of different implementations of active object-oriented database systems. Some of the most influential systems in this arena include HiPAC, Ode, Naos, Samos, and Sentinel, all of which have contributed to the active databases area and introduced many new concepts. The following is an overview of the former systems to give a sense of active OODBMS.

HiPAC [Chak89, Day96] was an early and substantially influential object-oriented active database project. The most important concept introduced by this project was the ECA (Event, Condition, and then Action) rule execution control model. The project defines three modes of event-condition binding, which apply to the condition-action binding as well. In the immediate mode, once the event occurs, the system immediately tests the condition. In the deferred mode, the condition evaluation is delayed until after the event occurrence and immediately before the commit point of the transaction. In the decoupled mode, the triggering transaction spawns a separate child transaction to evaluate the condition whenever the event occurs. The decoupled event-condition binding has decoupled-dependent and decoupled-independent modes. In the decoupled-dependent mode, the condition evaluation transaction commits only if the parent (event) transaction commits; otherwise, it rolls back if the parent transaction does. On the other hand, the decoupled-independent mode does not enforce such an allegiance relationship, and the condition evaluation transaction can commit or roll back regardless

of the parent transaction. The **Sentinel** project [Lee96] was a follow-on to the HiPAC project at the University of Florida. In the former, they developed a rule definition language called Snoop [Chak91, Chak93a, Chak93b]. The Sentinel project introduced several mechanisms for 1) monitoring events in a distributed fashion, 2) communication among applications using event mechanisms, and 3) integrating rules into a database programming language. Another follow-on project to HiPAC was **Reach** [Kud93], developed at the University of Darmstadt on top of OODB.

Ode [Geh91, Geh92a, Geh92b, Geh96] is another significant project that had an impact on the advancement of active databases. It extended the O++ object-oriented language (which is upward compatible to C++ with utilities suitable for DBMS application development) with facilities to define triggers and integrity constraints. Ode implements integrity constraints such that when a transaction touches an object, the rule system checks all the integrity constraints defined on that object. If the operation applied on the object by the transaction violates any constraints, then that transaction is aborted. The integrity constraints that are commonly enforced in this system are referential and value constraints. Ode is among the few database systems that provide some temporal query capabilities, such as order-based queries. For example retrieve the first instance, n^{th} instance, or the last. Furthermore, it offers limited temporal rule capabilities.

Naos [Coup94] was developed at the University of Grenoble. This project integrates a rule system into the O_2 object-oriented commercial DBMS. **Samos** [Gat91] is another European research product from the University of Zurich. It features a powerful and rich rule-definition language that gives the rule programmer a high ability to

define different kinds of rules. Naos uses an efficient Petri-net based event detection algorithm.

Object-oriented ADBMSs include AMOS [Sko96], Cactis [Hud89], ADAM [Dia91], Iris [Ris89], and others. The diversity in implementation methods and processing models used in the different object-oriented ADBMSs is due to lack of standards and recency of the object-oriented data model.

2.2 Temporal Database Systems

The aim of the projects discussed in the previous section was to develop an efficient active DBMS. Still, few ADBMS designs had thought of temporal issues. Among those were Ode and SAMOS, which included basic temporal order-based retrieval operators such as first, last, and n^{th}. However, these projects did not provide operators that are more elaborate. Here, we review some temporal database systems, most of which do not have a built-in rule system [Soo91]. Researchers have proposed several temporal database models and query languages [Gad88, Gor92, Wuu92, Snod93, Snod94a, Snod94b, Snod95, Tan93, Tan97], but our work has few common attributes with temporal databases. Hence, the focus here will be on consolidating aspects of temporal database systems that are directly related to this research.

Studies have produced various models for temporal databases. Most of the proposed models were basically temporal extensions of the relational model [Ari86, Tan86b, Clif87, Lor87, Sar87, Gad88, Gor92, Nav89, Snod90, Sar90a]. A general and complete description of a temporal database model is given in a study where the authors define a temporal database as a collection of state objects [Gal93]. Each state object is

represented by the tuple: **<*so-id, t_x, t_d, t_v, var-status, revised-so*>,** the elements of which are defined as

so-id: the state object unique id.
t_x: the time at which the value of the object was physically updated.
t_d: the decision time at which the value was decided (usually $t_x > t_d$ if available).
t_v : the valid time at which the value is valid in the real word. This may be an absolute time, an interval, or a set of intervals.
var-status: indicates whether the value is frozen (can be changed) or not.
revised-so: a set of so-ids of the same variable whose values are corrected by this state object.

The former representation of temporal databases is general, and many of the elements depicted in the ideal state object may be unnecessary for certain applications. In current commercial systems, only the id, the valid time, the transaction time, and the value are available. Most temporal databases follow some version of the previous model, which may lack some of the listed state object attributes. In general, temporal databases that maintain both the valid time and the transaction constitute the *bitemporal model*. Otherwise, the database will be classified as *unitemporal*, which only maintains either the valid or the transactional time stamp. Conceptually, throughout the rest of the dissertation there will not be any distinction between the formerly mentioned types of time stamping. In that sense, TriggerMan's temporal system uniformly treats information of the two models.

Time stamping the data is an essential issue in the process of maintaining historical data. The mainstream models for time stamping historical data are *interval time stamping* and *instant time stamping.* Interval time stamping [Sar87, Wied91, Dyr92] requires the database to add two time stamps for each row of a historical table. The two time stamps delimit an interval, which defines the period of time during which the

respective information was valid. Thus, each historical table maintains start and end timestamp columns (T_s, T_e). Contrariwise, instant time stamping requires only a single timestamp for each historical tuple. The later model assumes continuity of the data, meaning that the data will be valid until its next version's instantiation, and if it is the last version, then it is assumed valid until the current time.

Temporal query languages are typically SQL-based [Date93], and they differ from regular SQL in the functionality rather than the language constructs. TQUEL [Snod84, Snod93], HQUEL [Tan86a], TSQL [Nav87], TSQL2 [Snod92a, Snod94b, Snod95], and HSQL [Sar90a, Sar90b] are examples of temporal query languages. They provide the capability of composing historical queries such as getting the value of a record at some point of time, selecting the records that match a condition within a time window, and so on.

2.3 Active Temporal Database Systems (ATDBMS)

Several active temporal DBMSs have been developed as research projects; nevertheless, only a few of those have been commercialized, for example Postgres, which has been commercialized as Illustra and is currently part of Informix. This section briefly describes the most influential projects that substantially contributed to this field.

RapidBase [Wols96] is a research-originated commercial temporal main-memory DBMS. It was developed mainly for real-time industrial applications [Jok93]. This system was designed as a real-time active temporal database system that provides a limited capability language to define triggers on the data and perform other applicable operations.

Snoop [Chak91] is a rule definition language designed for Sentinel [Lee96]. In this language, the authors focus on classifying different types of events and on defining expressive temporal operators. Snoop is an expressive temporal language, though all the temporal expressions fit within the event definition clause of the rule. Like other temporal languages, this one involves some procedurality, where the user has to carefully combine the right operators and build event expressions to achieve what he wants. As examples of some interesting operators Snoop introduces the aperiodic (A, A*) operators which allow expressing an aperiodic occurrence of an event bounded by two arbitrary events. In addition, it provides the periodic operators (P, P*) which have an opposite action to the aperiodic operators'. The language provides tools that can cover a wide spectrum of temporal rules. Nevertheless, it is much more complex to use compared to SQL. Consequently, it requires a knowledgeable programmer or a highly skilled operator to use it for writing rules. Furthermore, it is not easy to understand the semantics of a rule without having to refer back to documentation.

FTL [Sist95a, Sist95b] stands for "future temporal logic", which is a temporal rule language. In this work, the authors introduce rather complex algorithms for implementing temporal rule condition testing. This language provides two basic temporal operators, which are "until" and "nexttime". Other temporal operators can be expressed in terms of the former two operators. Using global variables in combination with sequencing operators, many temporal events can be specified within the condition of a rule. Examples of rues written using the FTL language are given in chapter 7.

TREPL [Mot97] is yet another expressive and elaborate temporal aggregation language. However, in the author's opinion, it suffers from lack of readability and from

complex programming constructs. TREPL provides constructs for expressing simple and composite temporal aggregation. This language seems to be among the most expressive and complete temporal rule languages in the database research arena. Composite events in TREPL are expressed using regular expression-like operators as means to detect events such as immediate and eventual sequences, conjunctions, disjunctions, implicit recursion, and few other event expressions. TREPL is powerful and can fulfill the requirements of a wide range of applications that require intricate temporal aggregation in their rules. Examples of rules written using TREPL are given in a subsequent chapter.

Other active temporal languages include **PARDES** [Etz92a, Etz92b, Etz93a, Etz93b], **TALE** [Gal96], and so on. These languages are also powerful and complete. They follow models similar to those of aforementioned languages. Due to space constraints, we will not discuss any other temporal active languages.

Syntactic and semantic comparative analysis of the TriggerMan's language against others discussed earlier is presented in chapter eight. The comparative analysis comprises illustrative examples that are objectively selected to show how our language is superior to the others from different perspectives.

2.4 The Basic Structure of a Typical Temporal Rule Processing (TRP) System

Temporal rule systems are more general than regular rule systems because they handle both temporal and non-temporal conditions. Therefore, the issues of interest to classical rule systems are also present with temporal rule systems. The distinctive feature of TRP systems is their ability to process rules that involve the time dimension in their conditions. This is because time is a natural and vital element for complex applications.

Temporal rule systems are usually built as subsystems of temporal databases. Conversely, our temporal rule system is part of an autonomous asynchronous rule system that is separate from the database system. We believe that the main components of a typical TRP system are the following:

- Temporal condition composition utility (temporal sublanguage),
- Temporal condition testing logic,
- Timer manager, and
- Calendar manager.

In the first stage of the project, we focused on the first two components in order to produce a prototype of our TRP system. In more advanced stages, we intend to integrate Informix's calendar management capabilities with the system. Also, the overall operation of the TriggerMan TRP system will be enhanced and optimized.

2.5 The TriggerMan Project

A research group [Hans97] initiated the TriggerMan project at the University of Florida in 1996. The intention of the project was to design and implement a simple asynchronous trigger system. Such a module can be used as an external extension to any database system that lacks active capabilities.

The active capabilities provided by TriggerMan would enable the database system to process rules efficiently in an asynchronous fashion. The advantage of having an external rule processing logic is software independence. This means that if the rule system needs to be changed, there is no need to modify the database system holding the same data. Yet other advantages to such a system are 1) versatility of use, 2) the ability to

add active features to legacy databases, and 3) allowing centralized heterogeneous multi-source rule processing.

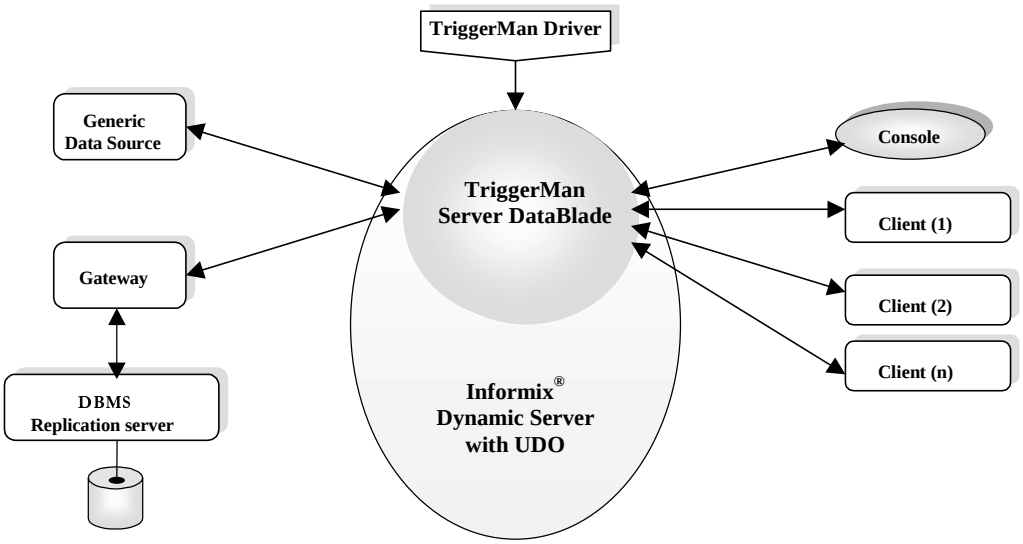

Figure 2-1: TriggerMan's architecture

Processing rules in an asynchronous fashion means that rules are applied to committed updates. This, in turn, implies that the system will detect integrity constraint violations after their occurrence. However, one advantage of asynchronous rule processing is the isolation of trigger processing from the triggering transaction. This results in a faster response time for update transactions, and provides the ability to define rules based on the outcome of multiple transactions. As shown in Figure 2-1, the architecture of TriggerMan consists of the following modules:

- **The TriggerMan server**: perform client and console request message handling, command parsing, condition matching, and rule action execution.
- **Console**: provides high privilege access rights and administrative features
- **Clients**: provide users with a user-friendly interface to TriggerMan.

- **Data sources and gateways**: communicate with external data sources, supply TriggerMan with replicas of updates, and allow remote access to needed information and data maintained by those sources.

2.5.1 TriggerMan Server Architecture

Currently, the TriggerMan server is implemented as an extension to a database system that acts as a storage and query processing backend. As will be discussed in upcoming subsections, Informix Dynamic Server with Universal Data Option (IDS/UDO) is the DBMS of choice to support TriggerMan's requirements. Therefore, TriggerMan will operate from within IDS/UDO, implicitly taking advantage of all its capabilities and processing power. Figure 2-1 depicts the general process architecture of the TriggerMan system. The process architecture of IDS/UDO–which inherently becomes the TriggerMan server's as well–is discussed in an upcoming section. In this architecture, TriggerMan is implemented as a DataBlade that extends IDS/UDO with TriggerMan server. The clients and the console interact with the server using a special API. A simple external driver invokes TriggerMan server periodically in order for it to carry out its task. The following chapter offers a more detailed discussion about the aforementioned architecture.

2.5.2 An Alternative Server architecture

An alternative to the previous architecture in which TriggerMan is tightly coupled to an extensible DBMS is a pure *stand-alone* system. In such a model, it is essential for the system to rely on its own efficient storage manager in order to achieve acceptable performance. Building an efficient storage manager is a quite complex task, which

requires an extensive effort. Even if TriggerMan relies on a traditional backend DBMS for storage and query processing purposes, then communication, data shipping, and query shipping overheads will slow the system and limit its performance. However, this architecture has the advantage of versatility, which enables it to operate with non-ORDBMSs. This is due to its independence and detachment from a backend host system.

In this architecture, the TriggerMan's server comprises several Virtual Processors (VProcs), which are similar to IDS's VPs. VProcs run on different architectures, and whenever possible, each VProc runs on a dedicated processor. On the other hand, all VProcs can run on one processor if resources are limited. At boot time, each TriggerMan VProc spawns and maintains the following threads:

- **Command server thread:** handles incoming commands and requests from client applications.
- **Matching thread:** receives update descriptors and tests them against any related active rules.
- **Rule action execution threads:** execute the matched (fired) rules actions.

Two application programming interface (API) libraries allow writing data source applications, and client applications. The console and other applications can use the client API to connect to the server, send commands and register for events. The data source API allows the programmer to write data sources and gateway programs for sending the update descriptors captured from their sources to TriggerMan server. Figure 2-2 shows the intended architectural platform, which is cluster of SMPs linked by a fast interconnection.

The above architecture achieves high versatility because it is not tightly coupled to any specific system. Even without having a local storage manager, the system can use ODBC [ODB94] as an interface to any mainstream DBMS to fulfill its storage requirements. Nevertheless, it suffers from high communication overhead if it is using an external DBMS for storage purposes. Furthermore, it requires a considerable effort to implement type and function extensibility, discrimination networks, and temporal constructs. These overheads are substantially reduced in the IDS/UDO-coupled architecture.

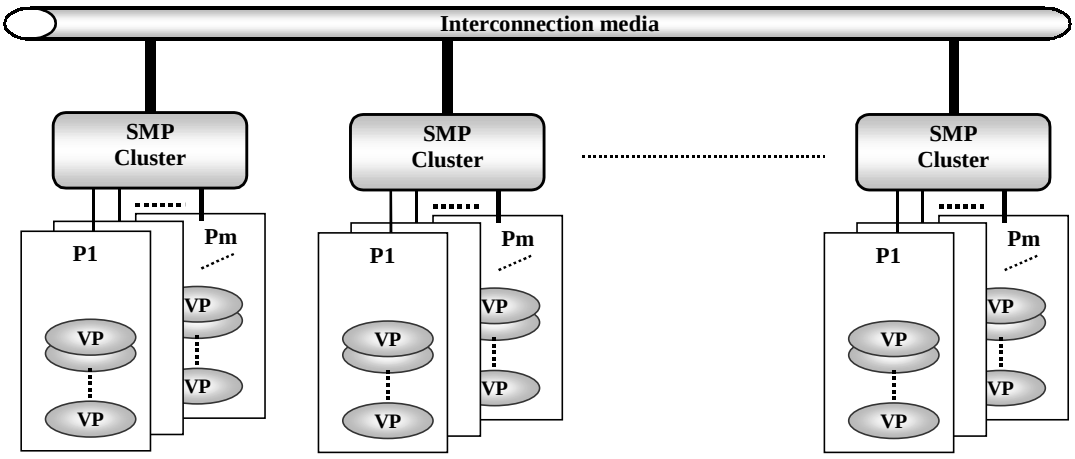

Figure 2-1: TriggerMan's intended architecture

2.6 Object-Relational Technology

Earlier in this chapter, we implicitly referred to both the relational [Codd70] and the object-oriented [Su83] data models. Although these were and still are the popular and most understood paradigms in the field of databases, a new hybrid data model called the object-relational model has recently emerged. This new paradigm promises high

performance, improved efficiency, and features that incorporate advantages of both basic models. Although this model is not commercially popular yet, many leaders in the area [Ston97] anticipate it to be the data model of the future.

Although mainstream commercial DBMSs have already utilized the object-relational model, their companies are still struggling to get the new idea across. Since these DBMSs involve a paradigm shift, it is normal for them to face some resistance in the beginning. Our project follows the object-relational model; accordingly, it was rational to use a matching database system as a backend for our storage management and query processing purposes. Illustratively, we have chosen a well-established commercial product that suits TriggerMan's requirements and provides superior features in many aspects such as extensibility, easy maintainability, etc. The Informix Dynamic Server with Universal Data Option (IDS/UDO) was the product of choice, though other commercial object-relational database systems (e.g. DB2/2) would have also fulfilled TriggerMan's requirements. The following subsections introduce IDS/UDO and discuss its different features and capabilities that TriggerMan utilizes. This will help the reader to understand some design decisions and implementation details that we discuss in succeeding chapters.

2.7 Informix Dynamic Server with Universal Data Option (IDS/UDO)

TriggerMan is designed to operate as an extension to Informix Dynamic Server with Universal Data Option. Hence, it is essential to discuss the capabilities and related modules of the host system. This section briefly introduces IDS/UDO, its features, and some related aspects.

IDS/UDO is an extensible object-relational DBMS. It supports complex data types such as images, audio, 3-D images, Web pages, and other complex objects. It is a structurally (new data types) and functionally (new functions) extensible system. The IDS/UDO's query language is SQL3-compatible [SQL3] with special commands for supporting extensibility.

2.7.1 Architecture and Capabilities of IDS/UDO

IDS/UDO is implemented as a client/server architecture that operates on top of a SMP system. On startup, several virtual processes (VPs) are spawned to take advantage of possible parallelism in the underlying platform. As shown in Figure 2-1, there are four types of processes. The following is a brief description:

- **CPU-VP**: maintains a number of non-blocking threads. Most of the IDS functionality is handled by CPU virtual processes. Threads running in these processes need to explicitly yield to allow others to share CPU cycles.
- **EXT-VP**: extension virtual process allows blocking threads to be spawned within its address space. This is usually used for special extensibility purposes by system users for CPU-intensive applications.
- **AIO-VP**: the asynchronous I/O processor handles asynchronous I/O independently and allows the thread/process that issued the I/O to carry on whenever possible.

- **NET-VP**: Network process handles network communication with clients and other servers.

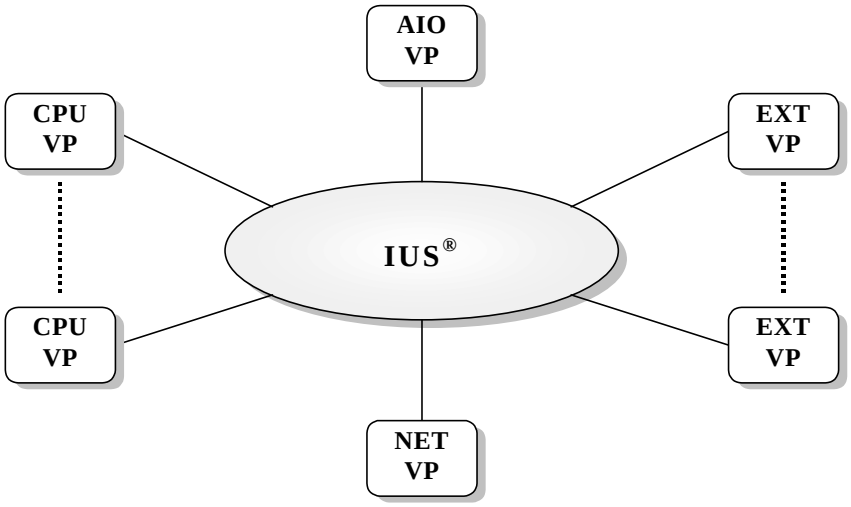

Figure 2-1: IDS/UDO's process architecture

IDS is highly scalable due to its symmetric multiprocessing nature. It also supports dynamic paralellizability where the system takes advantage of the available resources. It achieves high performance through raw disk management, effective memory management, dynamic thread allocation, and parallelism. Fast recovery, mirroring, point-in time restore, and data replication make IDS/UDO fault tolerant and highly available. Furthermore, it supports distributed database processing by allowing users to access different databases both locally and remotely.

2.7.2 Extensibility Features in IDS/UDO

IDS furnishes users with powerful extensibility tools that allow adding new data types and new functions to the system. Any extension to the IDS has to be in what is called a DataBlade module [DB97]. A DataBlade is a class of new object type or types

along with functions that may be applied to that class, or simply a set of new functions that take a list of arguments. A tool called *BladeSmith* allows easy creation of DataBlades through its template-based graphical user interface (GUI) and generates all the needed files. The output of the BladeSmith is pipelined into another tool called *BladePack*, which in turn generates some needed files and updates others accordingly. Afterwards, the DataBlade is ready to be installed by putting the different files in the proper directories. A *Blade Manager* tool allows insertion, deletion and manipulation of DataBlades once they are installed.

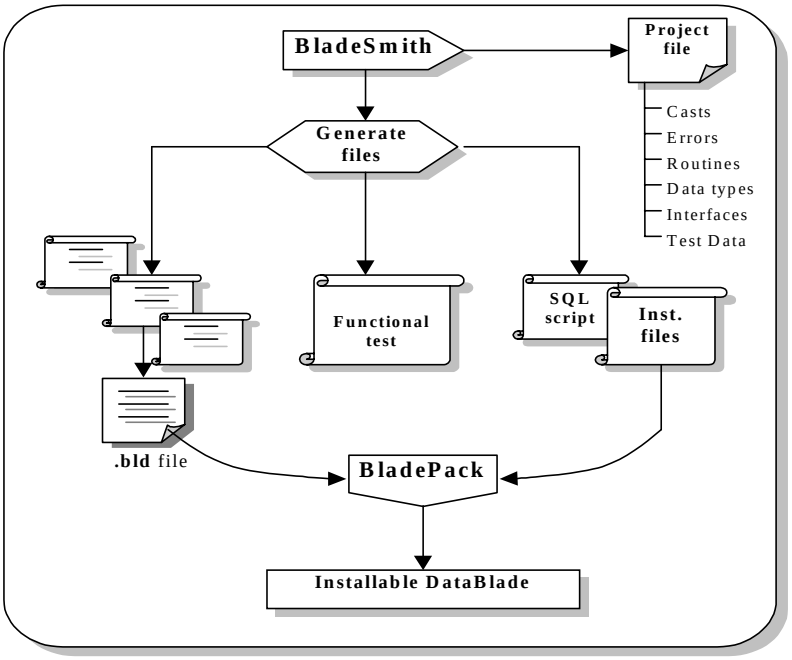

Figure 2-1: DataBlade development process (taken from [DB97])

Figure 2-4 shows the DataBlade development process up to delivering an installable DataBlade. IDS implements functional isomorphism at a higher level than the programming language, which is C or (Stored Procedure Language) SPL in this context.

The system maintains a function table in which each entry contains a function name, argument list, return type, and other attributes. This way, two functions may possess the same name as long as their argument lists differ. Adding a new function can be accomplished via a special SQL command designed for that purpose. All the argument types of the function must be either built-in or predefined (extended types) for the system to recognize them. As aforementioned, the name of the function may be repeated with a different argument list. It is important to note that functions are different from methods. Functions are not object-class-specific, but rather independent chunks of code that may take objects (data structures or variables) as arguments, perform a set of operations, and return either a result or nothing.

Complete extensibility details with respect to DataBlade composition, new function creation, and their integration with the system are complex. Due to space limitations, we have chosen to present a general overall view of the extensibility issue, avoiding meticulous and unnecessary low-level details. Throughout the rest of this dissertation, we focus on the main issues and high-level views, only when necessary; we discuss related low-level details.

CHAPTER 3
THE DESIGN AND IMPLEMENTATION OF TRIGGERMAN'S TEMPORAL RULE SYSTEM

The growth of the TriggerMan project ushered a proportional expansion of its original ambitions and goals. We realized that many of the modern and near-future applications deal with temporal data extensively. Such applications utilize various representations and formats to manage temporal data. The boom of computer usage in the world's economy during the past few decades has induced an exponential increase in the volume of operational data. This mandated a rapid advancement of information technology to cope with the requirements of new applications. One inevitable requirement of the evolving applications demands the consideration of the time dimension as a cardinal attribute of the data. The advancements in information technology have produced active database systems (ADBMS) which greatly reduced the complexity of applications performing monitoring and alerting. However, these systems adopted the traditional data model and overlooked the temporal attributes of data. The growing need for time recognition and utilization in active applications motivated us to consider augmenting the TriggerMan system with a temporal subsystem in order to handle general temporal rule processing. This chapter presents the major design issues and temporal features of TriggerMan. In this context, the focus is aimed at TriggerMan's Language, state maintenance, condition testing, and interaction with the host DBMS

system (that is, IDS/UDO). Other major system modules such as time-series and timers are discussed in separate chapters due to their importance and broad scope.

3.1 Temporal Trigger Language (TTL)

One main goal of this project is to provide a *declarative*, readable, and user-friendly trigger language, although it elevates implementation complexity. This arises from our strong belief in the concept that can be stated as: "whatever features and tools a system provides are never fully utilized unless they are easy to understand and employ." Hence, a substantial effort has been vested in making TriggerMan's rule language easy to understand, read, and use. Concurrently, it was important to maintain a simple high-level system model in order to control the complexity of development. Due to its declarative nature, TriggerMan's temporal rule language features flexible rule composition and provides a high level of abstraction. Therefore, the language allows users (analysts and application programmers with only limited training) to define a wide spectrum of temporal rules ranging from simple to complex. This section briefly introduces TriggerMan's rule language. The discussion is detailed and focused mainly on the temporal sub-language layout.

The temporal trigger language in TriggerMan is a natural extension of the system's basic triggering capabilities. The general format of a trigger in TriggerMan language is as follows:

create trigger <triggerName> [in <setName>]
from <dataSrcList>
[on <eventSpec>]
[start time <timePoint>]
[end time <timePoint>]
[calendar <calendarName>]
[when <condition>]
[having <havingCond>]
[group by <groupList>]
do [command | beginEndBlock]

As marked in bold in the above trigger creation syntax rule, the main pillars of the temporal trigger are the “on”, “start-time”, “end-time”, “calendar”, “having”, and “group by” clauses. Although it might not be obvious for the reader yet, the on-clause has a great impact on the semantics and behavior of the trigger according to how it is being deployed. Due to the aforementioned major clauses’ significance, the rest of this section is tailored around them. Other clauses will be mentioned as needed.

To illustrate the use of the trigger creation command, we provide the following set of temporal triggers examples. Since the intention here is merely to demonstrate the expressiveness of the language, the reader may ignore the trigger specification details for the time being.

Example(1): “Raise an event when the aggregate daily sales of the clothing department go over 5000”.

```
Sales (dept, item, qty, amount, ID#, disc)

create trigger highSales
from sales
on timer 1 day
start time #6/9/1997 8:00 AM#
when dept = “clothing”
having sum(qty*amount, “1 day”) > 5000
do raise event ClothesBigDay()
```

Example(2): “ Raise an event when the average daily sales volume of an item in a market basket is greater than 1000, and the total sales of the item are greater than 10 times the average daily sales of the same item”.

```
marketBasket(mbid,checkoutTime)
lineItem(lineno, mbid, itemcode, qty, unitPrice)
```

```
itemType(itemcode, description, price, category, avg_daily_sales,
avg_daily_volume)

create trigger marketBasketTrig
from marketBasket m, lineItem l, itemType t
on insert m
when m.mbid=i.mbid and i.itemcode = t.itemcode
having sum(qty*unitPrice, “1 day”) > 10*t.avg_daily_sales
and t.avg_daily_volume > 1000
group by t.itemcode, t.avg_daily_sales, t.avg_daily_volume
do raise event hugeJump(t.itemcode, Date())
```

TriggerMan’s temporal trigger system is inherently extensible. Hence, it can be extended from scratch with functions and data types that would suit the application’s requirements. Several continuous aggregates (temporal functions that use time-series) and incremental aggregates (temporal aggregates that do not need time-series) are fundamental. Functions such as “increase”, “decrease”, “average”, etc. are expected to be a potential constituent of a trigger’s temporal condition in many applications. Hence, it is rational to have a minimum built-in basic functionality that is ready to use without having the application programmer perform a nontrivial extension process. TriggerMan provides a set of built-in operators that we believe to be the most commonly understood and used. The built-in functions or incremental aggregates may not be a complete set of all possible temporal operators, but they are definitely useful and complete enough for numerous applications.

3.2 Built-in Temporal Functions

This section lists and demonstrates a number of built-in temporal functions that many applications may possibly use. A few conventions are clearly defined and used when describing temporal functions. The conventions appear in all of the functions for the sake of consistency.

Our built-in temporal functions are designed to take either all, or a subset of a standard set of arguments, where the brackets around an argument indicate optional parameter. The general format of our built-in temporal functions is as follows:

Function_name (**time-series**, **…**, [**window-start-time**], [**window-end-time**]), or

Function_name (**time-series**, **…**, **Offset**)

The following is a brief description of the standard arguments that are common to most of the built-in temporal functions:

time-series: a set of temporal instances contained in a unique time-series object. If the stand-alone architecture is used, then all the operational time-series are instances of our custom-built time-series class. Otherwise, in the IDS/UDO-coupled architecture, the system utilizes the time-series DataBlade object instances.

[window-start-time]: an optional beginning of the function's time window. If not provided, then it is assumed the trigger creation time, and is reset each time the trigger goes off. If the operator is preceded by an event in a sequence expression, then its start-time is set to the occurrence time of the previous event, and any given start-time is ignored. The format of the timestamp in TriggerMan's language is flexible and based on the American date/time standard. A timestamp consists of the following fields in the shown order:

 # Month *separator* Day *separator* Year [Hour] *separator* [Minute] *separator* [Seconds] [am/pm]#

'separator' can be any of the following symbols: - , /, : , and space.

This date/time format must be delimited from both sides by '#' to distinguish it from time intervals. The user is allowed to omit the whole time field, in which case the system assumes the time to be midnight (12:00:00.0000 am) of the given day. In

addition, time fields can be omitted in order, i.e. if a field is omitted then all the smaller granularity fields must be omitted as well. The day phase (i.e. AM, or PM) is optional, and if not available, the time is based on a 24-hour clock.

[window-end-time]: an optional end of the operator's time window. If not provided, it is assumed the timestamp of the last update applied to the time-series, and has the same formats and rules as the start-time.

Offset: an offset from the last point in the time-series, the time window in this case starts from (last time stamp – offset) to the last time stamp. This offset substitutes the time-window delimited by start-time and end-time.

Exp.: a Boolean expression involving the result of the continuous/incremental aggregate. The standard operators include: >, <, =, !=, *, etc. Table 3-1 lists a sample of TriggerMan's built-in temporal functions, such that examples of the listed functions usage are given in a subsequent chapter.

3.3 Nested Temporal Functions

Nesting of functions adds flexibility and expressiveness to the condition of a temporal trigger. Conversely, it potentially elevates the level of complexity and confusion. In some cases, multi-level nesting makes the function ambiguous and hard to conceive. For example, the multi-level nesting implemented as part of INGRES® incurred high complexity, and the semantics of some of their nested operators were ambiguous [Eps79]. After reviewing different real-life applications, single-level nesting proved to be the prevailing depth of needed nesting in most cases. Therefore, we restricted the degree of nesting in our language to a single level.

Table 3-1: Selected built-in temporal functions

Function	Return Type	Description
Increase	Boolean	Detects absolute or relative increase within a window.
Decrease	Boolean	Detects absolute or relative decrease within a window.
Occur	Boolean	Checks if a given expression is true within a time window.
StayInRange	Boolean	Checks if a variable has stayed within a given range for some duration.
Difference	Boolean	Detect absolute or relative difference within a window.
Holds	Boolean	Detects a steady state of a variable within a window.
UpdateCount	Scalar	Returns number of updates to the given variable.
Sum	Scalar	Calculates the sum of a variable's instances within a window.
Average	Scalar	Calculates the average of numeric value within a window.
ExpAverage	Scalar	Calculates the exponential average within a window.
Maximum	Scalar	Returns the maximum value of a variable within a window.
Minimum	Scalar	Returns the minimum value of a variable within a window.
PhaseCount	Scalar	Counts number of (increase,decrease, no change) phases of a variable.
MaxIncrease	Scalar	Reutrns the maximum increase of a variable within a window.
MaxDecrease	Scalar	Returns maximum decrease of a variable within a window.

The advantages of limited nesting are the addition of more expressiveness and flexibility to the language. Single-level nesting also avoids the confusion and complexity associated with multi-level nesting. The TriggerMan temporal language permits only one level of nesting. Namely, the first argument of a Boolean temporal operator can be either a time-series (no nesting), or a *flat* temporal aggregate. One-level-nesting facilitates the composition of triggers that monitor the variation of sums or averages, or in general, monitor the variation of temporal functions over a period of time. The result of the internal temporal function is represented as a time-series object. Thereupon, the condition testing logic recursively applies the external temporal function to the resulting (derived) time-series.

In order to adopt a single level nesting, the condition checking process has to be slightly different. In case of a nested temporal function, the condition checking process is done in two steps. First, the inner-level temporal function is applied to the time-series within the given time window. Since the result of the previous temporal function application is a derived time-series, the next step is applying the outer-level operator to this time-series. A nested temporal operator has the following syntax:

Temporal-operator (Aggregate–temporal-operator, "expr", [start_time], [end_time])

An example of a nested function would be calculating the monthly average of daily sales. Assuming the sales table to contain only individual checkout records, the internal function aggregates the daily sales in a new time-series. Thereupon, the external function calculates the monthly average for the resulting time-series. Figure 3-1 demonstrates the steps taken to evaluate the following function: **Avg(sum(sales, "1 day"), "1 month").**

The first step evaluates the inner aggregate, which generates a time-series that holds the sum of the daily sales. The second step applies the average function to the daily sum time-series (derived time-series) and returns the average daily sales for the previous month.

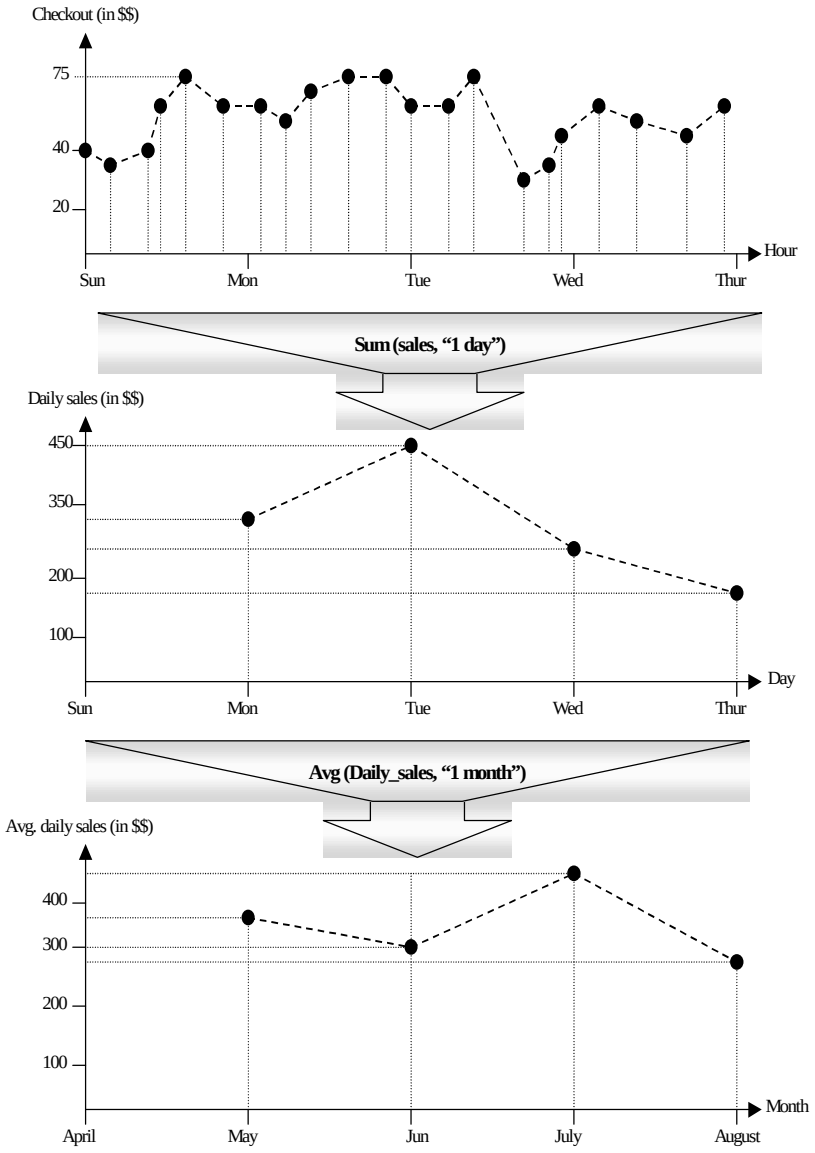

Figure 3-1: An example of a nested temporal operation

3.4 Expressing Simple Temporal Sequences

In the literature, most of the temporal active systems have provided tools for expressing sequences of events in the form of regular expression-like formats [Chak91, Geh92b, Sesh94, Sesh95a, Sesh95b]. Expressing a sequence of events is not trivial and some times requires a deep understanding of both the events themselves, and the language needed to express them. Many useful sequencing operators have been proposed, among which are after, before, immediately after, eventually, etc. Triggers with event sequences involve extensive processing and sometimes suffer from the complexity that might be associated with detecting the sequence. In most sequence detection algorithms, a state has to be preserved in order to detect the sequence occurrence correctly. Therefore, regular expression evaluators are the most suitable approach for dealing with this problem.

Conversely, a trigger that has a sequence of events as part of its temporal condition can be decomposed into a number of triggers that is equal to the number of events in the sequence. To achieve the same effect as the original trigger, the composite sequence triggers have to detect the sequence occurrence when it happens. Figure 3-1 shows an example of the simple decomposition process. In this technique, each event in the original sequence trigger is placed in a separate trigger definition. If the event is not the last in the sequence, then the corresponding trigger's action activates the next trigger down the sequence chain. Otherwise, the action executes the overall action of the event sequence. Once the last trigger in the sequence chain fires, all the previous events in the sequence are guaranteed to have occurred in the proper order.

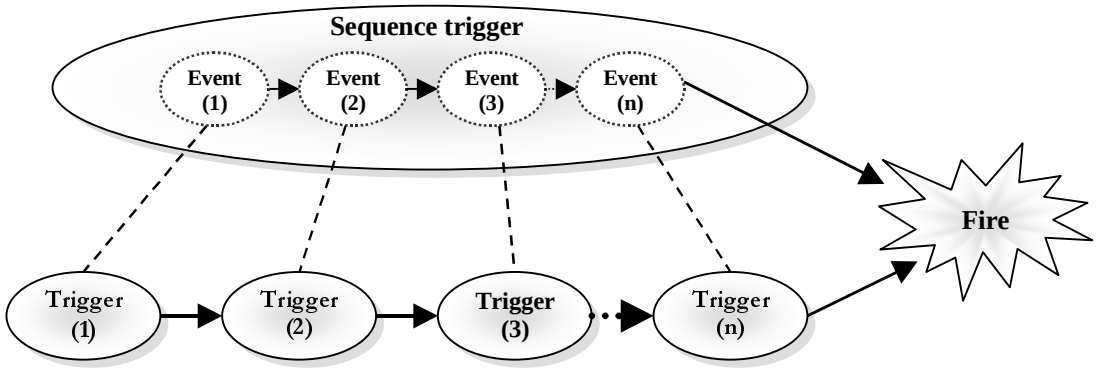

Figure 3-1: Expressing simple event sequences using event decomposition technique

3.5 Expressing Complex Temporal Sequences

Complex temporal sequences are regular expression-like where recursive events are specified within the expression. For example, the following temporal sequence: "if a stock's closing price *rises by 20%* to a new value P, *then* report the stock-closing price *every day* as long as it stays above P" cannot be expressed using the trigger cascading method. Such a temporal sequence actually consists of two consecutive temporal events. The first is a simple event, which is "a 20% raise in the original stock-closing price to a new value P". The second event is a recursive event, which is a "the stock-closing price does not drop below P during one day". In this example, the system has to first detect the first event, and once that happens, the system checks for the second event every day. If the second event occurs, the system has to execute the proper action, and recursively keep checking the same event. Otherwise, i.e. the second event does not occur, the system resets its state, and starts all over again by testing for the first event of the sequence.

The above discussion is simply a regular-expression evaluation. Hence, complex temporal events can be detected using a regular-expression evaluator which takes a

specific alphabet and a set of operators. Since TriggerMan is extensible, users can extend the temporal functions to include a regular-expression evaluator (function) that would fit their applications. An example of a useful regular-expression evaluator could be *Reg_Expr_Eval*. This evaluator would take all TriggerMan's built-in functions as an alphabet, and a set of operators that include: "immediately: →" and "eventually→→". Accordingly, the temporal sequence in the former stock-closing example can be expressed as follows:

Reg_Expr_Eval (*increase*(stock_price,"20%", "1 day") → *noDrop*(stock_closing))

The regular-expression evaluator can be as general as needed. Also, the complexity of such module can vary depending on the implementation. It is our intention to include a general regular-expression evaluator in an advanced version of TriggerMan.

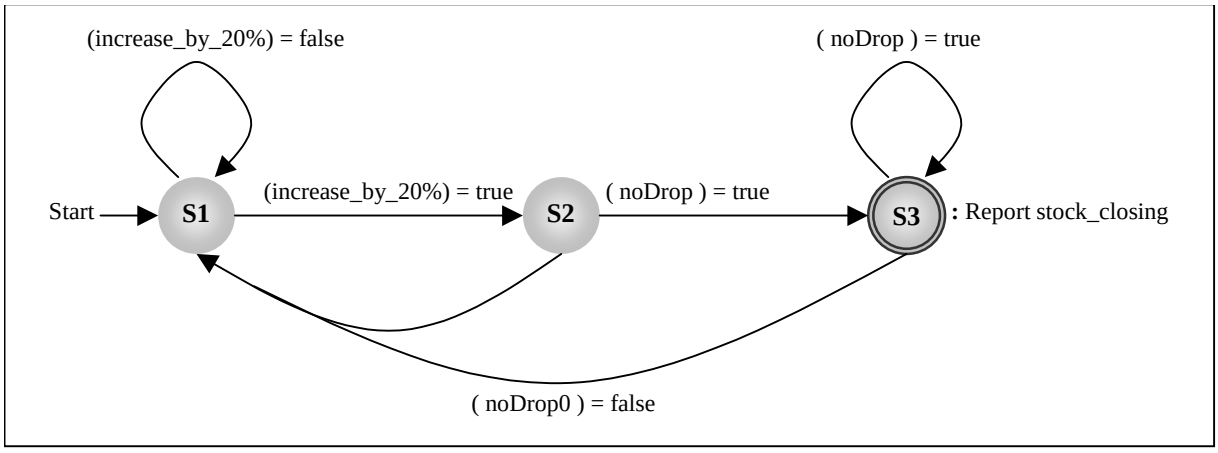

Figure 3-1: The automaton for the stock_closing example

3.6 Adding New Temporal Functions

Since TriggerMan is extensible, adding new continuous and incremental aggregate functions can be achieved via special extensibility tools. TriggerMan's built-in temporal functions are a minimum set of the possible complete temporal operator set. Using the available extensibility tools, users can augment the temporal system with their own temporal functions as needed. However, the extension process is not trivial and requires knowledge about the procedure through which new functions can be added to the system. IDS/UDO provides a specific procedure to add new functions and data types. A similar procedure applies to adding temporal functions except for a slight difference. Temporal functions are required to possess specific attributes in order to carry out the correct operation. We provide templates for temporal functions and incremental aggregates, which automatically include needed structures and logic. Users must use these templates to plug in their code in order to guarantee the correct operation. Once the temporal function template is filled and debugged, an extension procedure allows the incorporation of this new function within the system. Namely, once a new function is in the final form, it has to be explicitly registered with the system using a special command designed for that purpose. Thereupon, users may call the function in any trigger condition or action, provided they pass the correct set of arguments. Once a function is added to the system, it becomes a persistent part of the system until it is explicitly dropped. A more detailed discussion of adding temporal functions is incorporated within section 3.8.

3.7 Temporal Condition Testing

Condition testing of triggers in TriggerMan is achieved via discrimination networks. These are dynamic, efficient, and highly parallelizable data structures that allow multi-table condition testing even with high update frequencies. There are several types of discrimination networks. Examples of some of the most popular networks are *Rete* [Forg82] and *Treat* [Mir87]. TriggerMan uses an optimized hybrid of former two networks called *Gator* [Hans96b].

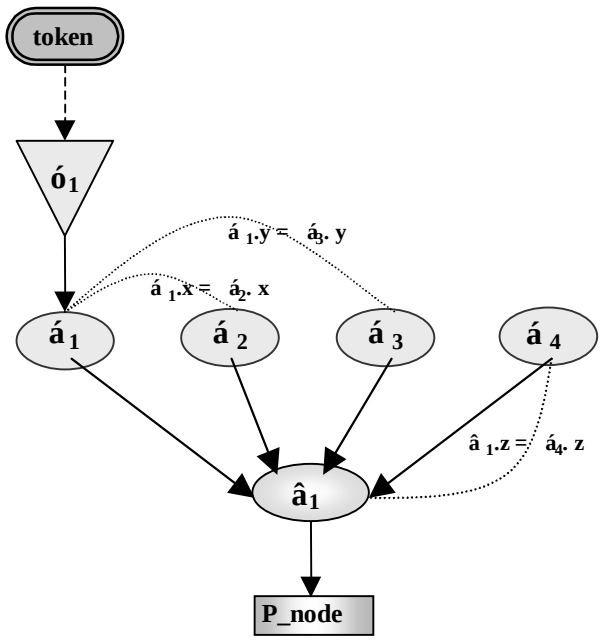

Figure 3-1: An example of a Gator discrimination network

Gator networks represent each rule as a rule-condition graph that is equivalent to a query graph. It maintains a data structure called • –node for each tuple variable and an edge for each join in the rule condition. If available, a selection predicate node called a • –node is connected on top of the corresponding • –node. The Gator network is

optimized for each rule such that it contains a tree of nodes, where selection predicate nodes are the leaves, and the root is a special node called the P–node. The join result of multiple • –nodes may be materialized in a data structure called a • –node. The P-node takes place of a • –node at the bottom of the network and collects the final result of the condition testing as described below. When a change in the database occurs, depending on whether it is an insert, a delete, or an update, a token is generated and fed to the proper • –node through its • –node if there is one.

When an • –node receives a token, according to the operation, it either stores it or deletes it, then joins it to any other node that has a join edge connected to it. The join results are stored in or deleted from the • –nodes underneath the • –node. In case of an insert or an update, if any joined tuple filters through the network into the P–node, then the corresponding token must have matched the rule condition. Hence, the rule fires, and its action has to be executed as soon as possible. The previous algorithm is used for testing the condition of non-temporal rules. For temporal rules, there is an additional step for testing the temporal condition of the respective TriggerMan rule before it fires.

Most expressions in the temporal condition of a trigger contain temporal aggregate function(s). Some temporal functions require time-series to operate on; thus, the TriggerMan server maintains such time-series. At trigger's condition-evaluation time, the server applies a respective temporal function to a corresponding time-series. Other temporal function need only the old value of a variable, therefore, their evaluation does not involve any time-series. If the architecture is stand-alone, then the time-series used is an instance of our own custom-designed time-series class. On the other hand, if the architecture is IDS/UDO-coupled, then the time-series would be an instance of the

IDS/UDO time-series DataBlade object. Time-series in the condition is either implicit or explicit as described in the following chapter.

Condition evaluation for both temporal and non-temporal conditions in a rule is stack-machine based. Condition expressions are interpreted into an intermediate code (p-code), which is eventually executed by the stack machine. When the expression evaluator encounters a temporal function call, it pops it, invokes it, and pushes the result back when returned.

As described in the timer system discussion, the semantics of the when-clause change when the trigger is timer-driven. Namely, the "when" clause acts as a filter for the time-series contained within the temporal condition of the trigger rather than qualifying the trigger to fire. Only when the timer goes off does the system test the temporal condition of the trigger and carry out the proper action. Thus, the discrimination network of a timer-driven trigger acts a little bit differently. Unlike regular discrimination networks, whenever a tuple set falls into its P–node, the action becomes restricted to updating the related time-series and no further. In order to achieve correct operation, the system draws a clear distinction between rule discrimination networks and filter discrimination networks.

3.8 Testing Continuous Aggregate Functions

We refer to temporal aggregates that require a time-series to store their state as continuous aggregates. Thus, a continuous aggregate function takes a time-series as one of its arguments in order to access the history upon which it is to operate. A continuous aggregate may use an explicit or an implicit time-series. As described in the following

chapter, an explicit time-series is a global structure that may be accessed by multiple triggers. On the other hand, an implicit time-series is trigger-specific and its existence is solely dependent on its host trigger's. When the command server encounters a continuous aggregate function, it first extracts the name of the time-series from the argument list. Afterwards, the command server searches the explicit time-series catalog for an existing time-series with the given name. If it finds one, then the command server uses that explicit time-series whenever the function is invoked. Otherwise, i.e. the server does not find an explicit time-series, the command server validates the time-series attribute; thereafter, it creates an implicit time-series to be used whenever the function is invoked within the respective trigger. At testing time, explicit time-series are protected using semaphores because they are sharable and are critical sections in parallel environments. When a trigger is dropped, the system deletes all implicit time-series that have been created for its continuous aggregates. Figure 3-5 illustrates the procedure used by the temporal system to locate or create a time-series referenced by a continuous aggregate.

A trigger's condition consists of two sub-conditions, the non-temporal condition and the temporal condition. Since the temporal condition always comes after its counterpart, its evaluation is dependent on the result of the non-temporal condition. If the non-temporal condition is absent, then it is always assumed true and the system immediately tests the temporal condition when the trigger's condition is to be tested. TriggerMan uses a stack machine for evaluating expressions contained within a trigger's condition.

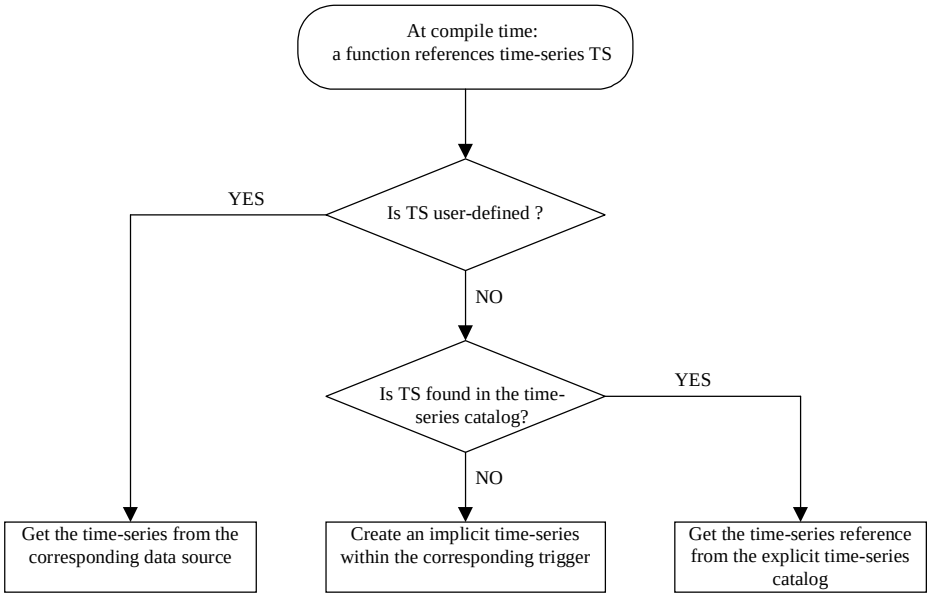

Figure 3-1: Applying a continuous aggregate to a time-series

3.9 Testing Incremental Aggregate Functions

Incremental aggregate functions do not require a time-series to maintain their histories. For example, an aggregate like sum requires only the previous sum in order to calculate the new sum when a new data item is inserted, or an old item is updated or deleted. TriggerMan allows the usage of incremental aggregates only in the “having” clause because we consider these aggregates as special temporal functions (they need the previous state for recalculation). During the parsing stage, the parser assigns each function in the “having” clause a unique identifier local to the trigger as shown in figure 3-6. This identifier is later used to identify the function at the time of invocation. This is necessary because multiple instances of the same function may be present in the condition. Users must use an aggregate function template to add their custom-made functions in order for the system to properly invoke them when employed in a trigger.

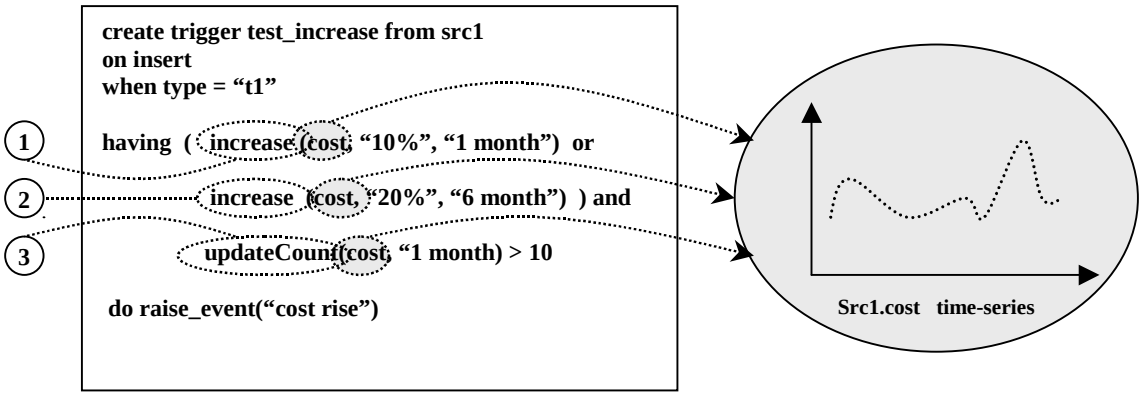

Figure 3-1: Numbering temporal functions and creating implicit time-series

When a trigger is created, it invokes all the constructors of any state objects used within the temporal condition. The constructor of a state object simply creates a single-row state table and initializes it. If a trigger is dropped, then the destructors of all state objects are invoked, which in turn destroy all corresponding state tables. Figure 3-2 shows the templates for the state object constructor and destructor.

```
Type_state_construct ( state_table_name )
{
      Create the state object table:
     EXEC SQL{
                    create table state_table_name
                            attr1: state_obj_type
   //              }
  }
```

```
Type_state_destruct ( state_table_name )
{
    // Drop the state object table:
    EXEC SQL{
                    drop table state_table_name
                }
}
```

Figure 3-2: The template for the state table's constructor and destructor

The name of a state table is a concatenation of the trigger's name and the aggregate's unique identifier (assigned at parse time). Since the trigger's name is unique

system wide and the aggregate's identifier is locally unique, then the table will always have a unique name both locally and globally. The template of an aggregate function is shown in Figure 3-8.

As the system attempts to test the temporal condition of a trigger, it first evaluates all the aggregate functions by applying what is called local argument substitution. This is accomplished by inserting the local values of the trigger name and the aggregate identifier into the argument list of the aggregate function's template. Figure 3-9 illustrates the local argument substitution technique for a simple aggregate function.

```
Return_type Agg_func_name (state_table)
{
  State_obj_type old_value, new_value
 // get the old value from the state table:
EXEC SQL{
                    select value into old_value
                    from state_table
            }
.
.
User code
.
.
// Update the state table with the new value:
EXEC SQL{
                  update state_table
                   set value = new_value
            }
```

Figure 3-3: The template for an Incremental Aggregate function

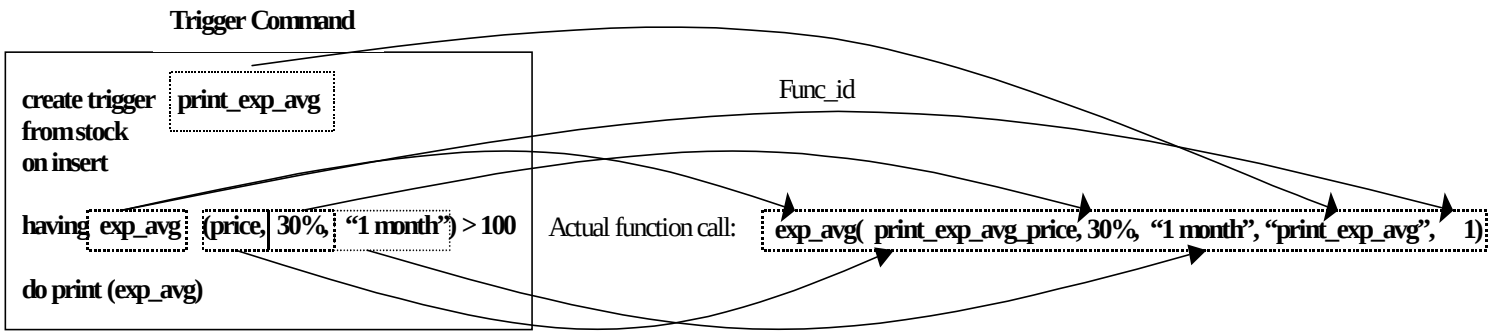

Figure 3-4: Example of local argument substitution

3.10 TriggerMan's Interaction with IDS/UDO

TriggerMan is designed to operate as an extension to IDS/UDO, though, the later is actually used as a backend storage and query processing system. Hence, TriggerMan is implemented as an IDS/UDO DataBlade, taking advantage of its architectural features, extensibility, parallelism, and storage capabilities. As shown in Figure 2-1, A simple driver controls the TriggerMan server by periodically invoking a function allowing it to cooperatively gain a share of the CPU cycles to test its rules and take proper actions accordingly. This technique simulates a server thread that shares CPU time with other threads in the system. The reason for such an approach (simulating rather than creating TriggerMan threads) is the complexity and the risk associated with creating new threads within IDS/UDO. These are actually implementation specific issues that relate to the system portability and they do not affect the high-level concepts.

Since IDS/UDO provides powerful extensibility tools, TriggerMan makes use of these tools and hence inherits the extensible nature of the host system. Therefore, temporal functions may be added in a specific procedure and become immediately usable afterwards, like other regular functions in IDS/UDO. Furthermore, time-series in

TriggerMan are actually instants of the time-series DataBlade's object class. IDS/UDO supports functional polymorphism in the sense that the function name, in addition to its argument types, uniquely distinguishes it from other available functions in the system. Consequently, there is no limit on the number of functions with the same name as long as they differ in their argument lists. Once a temporal function is added to the set of user defined functions, it immediately becomes visible to TriggerMan and to all other system DataBlades and applications. Thereafter, users can imbed a call to that function within their temporal condition in a trigger definition.

When clients use TriggerMan's language to define their triggers, a command server parses the command string, type checks it, then executes the resulting syntax tree. Throughout the command processing phase, internal data structures (and possibly tables) within IDS/UDO may be created and/or updated. Eventually, the new trigger is added to the list of available triggers in the system. The system treats other commands the same way, but the degree of interaction with the internal data structures and tables may vary.

3.11 Using Calendars to Control Rule Activation

Calendars are important especially for business rules. In many practical applications, the simplest role of calendars is to divide the time line into on/off intervals. Off-intervals define the times during which the related activity is either stopped or invalid, such as holidays, breaks, blackouts, etc. Oppositely, on-intervals are the rest of the time line intervals during which of the related activity operates (or is expected to operate) normally. TriggerMan allows the use of any particular defined calendar to

participate in controlling the activation of a trigger. Calendars can be created using the Informix time-series DataBlade's interface.

We define the valid active time of a trigger to be the set of intervals during which the trigger responds to related events and fires if its conditions are matched. Each rule has start and end times; these are either defined by the user, or set to default values as discussed earlier in this chapter. Hence, the valid active times of a rule are simply the result of time-aligned intersection of the calendar definition and the interval delimited by the start and end times as shown in Figure 3-10. If a calendar is not used, then the interval bounded by the start and end times defines the valid active time of the trigger.

The conjunction of calendars and trigger activation control clauses (start and end) give users the freedom to define a trigger's activation time precisely. This way, a trigger is guaranteed to operate as desired and when desired. During a trigger's inactive times, its internal structure and state may change, but the system hides all the events from it, therefore the trigger becomes event-blind and unable to fire. An example of such a situation is a trigger that has a function application on a time-series. Time-series collect qualified updates at all times. Even if the time-series is trigger-specific, it responds to updates even during the corresponding trigger's inactive times. The following chapter offers detailed discussions of all issues related to time-series.

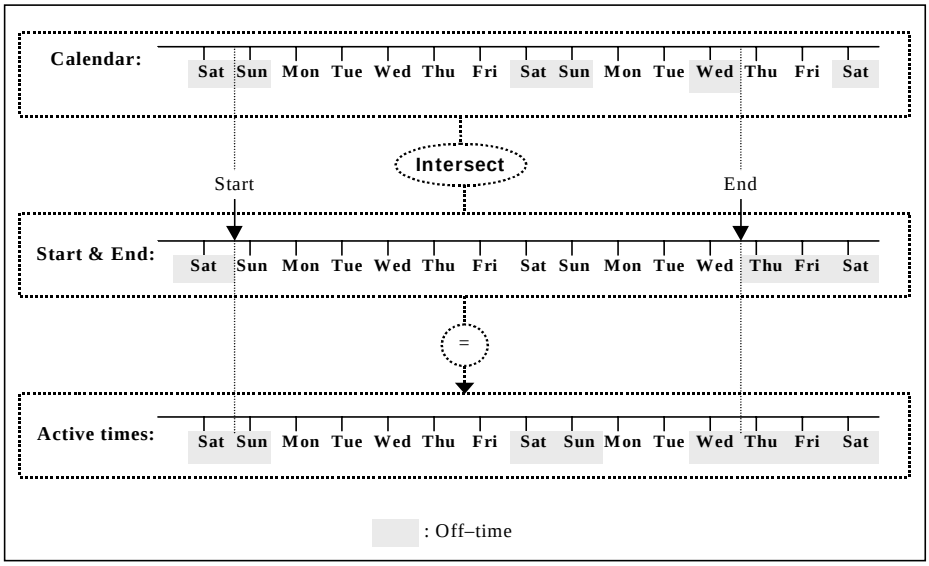

Figure 3-1: Calculation of trigger's active intervals

CHAPTER 4
USING TIME-SERIES TO REPRESENT TEMPORAL VARIABLE HISTORIES

This chapter presents the temporal data maintenance technique. TriggerMan uses time-series for maintaining historical data because of their ease of use, versatility, and compatibility with many temporal applications. Here, we explore the different types of our time-series, the related language commands, and some high-level details.

4.1 Volatile Time-Series

Any temporal system must use some structure to store temporal data, which usually consists of two components: timestamp and data. Different temporal systems use different representations for temporal data. In our case, we chose the time-series representation, which was originally used for statistical analysis and forecasting [And75, Box76, Bro87, Shu88, Mil90], but later became a very popular representation of temporal data. There are two types of time-series categorized according to their durability; the first is the volatile time-series, which, as the name indicates, vanishes as soon as the system is brought down, regardless of the cause. The second type is the persistent time-series, which is supposed to endure any failure or normal shutdown of the system. The following two subsections describe in detail the design and implementation of these two time-series representations. In addition, an overview of Informix's time-series DataBlade is presented since their time-series is used in TriggerMan's Informix-coupled configuration.

The time-series in our model is a simple organization (doubly linked list) of a tuple (or state object) collection. More complex techniques for time-series representation such as “the time index” can be used to maintain time-series and achieve fast access to their contents [Elm90]. The index key is the transactional or valid time of the tuple. The transactional time refers to the commit time of the transaction that produced the tuple. We consider the valid time to be the time TriggerMan server becomes aware of a tuple if it did not have a time stamp associated with it when it was received. Although, other researchers have a different definition of valid time, but the former definition is used to guarantee the operation of our temporal system even when plugged into a non-temporal DBMS. Each time-series is simply an index on the time stamp of history elements. Each of those elements consists of a time stamp structure, and a pointer to a tuple in the corresponding tuple collection as shown in figure 4-1.

There are two types of time-series with respect to initialization; one is the primed time-series, which is usually loaded from a persistent copy (file or table). The other is what we call on-line time-series which starts empty (of course, the corresponding tuple collection will be empty as well), and gets updated as tuples arrive from the source database and inserted into the proper tuple collection. When a time-series is on-line, we assume the history to have started at the time of the time-series creation instant, which is the same as the corresponding trigger’s creation time.

Our time-series provides a simple and useful interface for external modules to analyze its contents. An external application can apply several functions to a time-series such as setCursor( ), getNext( ), countElements( ), etc., however, its contents are read-only under user mode, which means they are non-updateable from user point of view.

Update interface methods are available only within the system. Examples of such methods are insert( ), trim( ) and some others. In case of using Informix's time-series, the interface is similar with a much wider variety of functions that can be used to operate on the time-series.

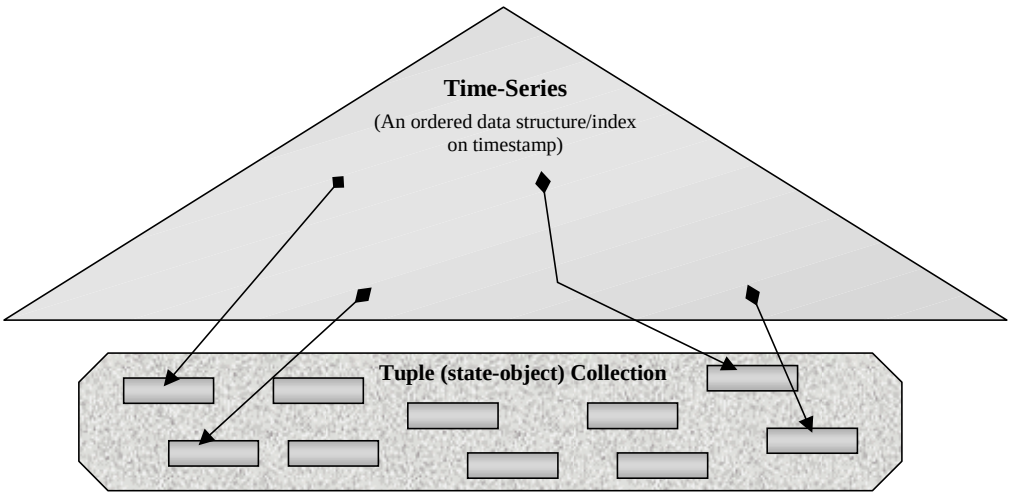

Figure 4-1: Time-series structure

4.2 Persistent Time-Series

Actually, when a time-series is created as persistent, the only difference from it being volatile is that it guarantees the persistence of any new update applied to its tuple collection. Moreover, it should be able to initialize (load) itself from its tuple collection when the system is started. Other than that, it behaves exactly like the volatile time-series, and supports the same interface. Most of the time-series used in our system will be persistent by default unless the user specifies otherwise. When a temporal trigger is defined, the temporal system analyzes each valid temporal function. If the function is a continuous aggregate that uses a temporal attribute or object (we call it a temporal variable), then it first tries to find a matching explicit time-series. Explicit time-series are

created by users, as we will describe in the coming section. If a matching time-series is found, then that time-series is used to evaluate the corresponding function. Otherwise, the system will implicitly create a time-series to track the history of the temporal variable, in order for the continuous aggregate function to be correctly evaluated.

4.3 Informix Time-Series DataBlade

Since one version of TriggerMan is built as an extension to the Informix DBMS as described earlier, it is sensible to take advantage of the host system's native capabilities and use their time-series DataBlade since it offers the required functionality. A DataBlade may add new data object(s) and procedures or functions that take argument lists and return familiar data types.

The Informix time-series DataBlade [TSDB97] consists of two major modules (data objects) that are highly versatile and programmable. The first is the time-series it self, which is simply a set of time stamped objects called elements. Each element is a row data type, which is an extended data type that has a set of attributes. They divide their time-series into two types: regular and non-regular. A regular time-series is a time-series with a fixed time-interval temporally separating any two elements in it. This type of time-series may be temporally organized using time stamps, or simply an offset from an origin point, which is the time at which the time-series starts collecting or accepting new data (elements).

Non-regular time-series are slightly different because as the name indicates, there is no restriction on the interval separating any two elements. Hence, they are suitable for applications where the data points are temporally unpredictable. In this case, timestamps

are the only temporal ordering tool. Otherwise, both types of time-series are treated uniformly, and the same methods that apply to one type apply to the other.

The other major module or data object is the calendar. Informix time-series DataBlade provides a calendar object that can be created and customized to fit the users temporal inclusions and exclusions to control data flow into a time-series. In temporal database terms, calendars are used to define the validity of the data [Soo92, Chan94]. For example, the business calendar defines all days except Saturday and Sunday as the valid interval of a week. In other words, only the data received in the valid interval may be taken into consideration and stored in the corresponding time-series. This way the user can reduce the amount of redundant information flowing into the time-series and save the time to identify and eliminate such information in the analysis phase. Although invalid data may be useless with respect to analysis, it might be needed for other purposes. The Informix extended calendric system allows the storage of invalid information and hiding it from analysis methods.

A set of mathematical and analytical functions is provided with the time-series DataBlade. Furthermore, the user may extend the time-series with his/her own functions through SQL or API interfaces. Therefore, Informix's DBMS guarantees the completeness of the solution with respect to any application via extensibility tools. Inherently, extensibility is a powerful feature of TriggerMan's as well.

From the former discussion, the reader can realize how elaborate and general the Informix time-series is. In view of that, we are convinced that using their time-series is complete enough for most practical applications. Utilizing such capability will not only save a considerable amount of development and implementation time, but would also

empower TriggerMan's temporal system with a solid and versatile historical tracking tool. Throughout the following sections, whenever time-series or calendar is mentioned, it is implicitly referring to the Informix's time-series DataBlade objects.

4.4 Implicit Time-Series

When the having clause of a trigger contains a continuous temporal aggregate that operates on a time-varying attribute, then the system automatically creates a time-series to keep track of its history. As the name indicates, such time-series are created implicitly and are invisible to the user. Implicit time-series are trigger-specific, in that sense, their existence is entirely dependent on the existence of the corresponding trigger. A trigger may posses multiple time-series that track several temporal attributes. Once created, an implicit time-series maintains the data within a time window that is equal to the maximum relevant temporal function's specified window. For example, for the temporal function: increase(quantity, "10%", "1 month"), an implicit time-series will be built on the "quantity" attribute, and will keep 1 month worth of quantity values. Within the same having clause, if another function uses the same time-series but has a 1-year time window, then the time series has to keep a year's worth of relevant data. Such time-series are updated on-line with values extracted from update descriptors that successfully filter through the when clause condition of their trigger. Once a user drops a trigger, all of its implicit (private) time-series are immediately deleted. The intent of implicit time-series is to make it easier on the user to write temporal triggers because they don't have to worry about the creation of time-series to track the historical attributes used in the continuous aggregate. An example of implicit time-series creation is as follows:

" Create a trigger that would increase the maximum room discount in a hotel with the given name and has a rating that is less than three stars when the average weekly occupancy drops by 20%:"

```
create trigger HotelDiscount from hotels
on timer 1 month
when hotels.name = "DiscomfortInn"  and  hotels.rating <= 3
having decrease(avg(occupancy, "1 week"), "20%", "1 month")
do exec increase_discount_by_10%
```

In the former example, the temporal system first makes sure that there is an attribute called "occupancy" in the hotels table (data source) while it is validating the trigger definition. Then it creates an implicit time-series to track the occupancy attribute (assuming there is no explicit time-series for tracking the occupancy). As described in the previous chapter (see figure 3.5), the priority is always given to explicit time-series. This means that the system tries first to locate an explicit time-series that is defined on the "occupancy" attribute (with the same selection predicate) for instance, and uses it if available. Otherwise, the system creates an implicit time-series. This allows the system to exploit any possibility for sharing a time-series, in turn, reducing the storage requirements when the average time-series size is large.

4.5 Explicit Time-Series

Explicit time-series are created by the user or administrator via commands designed for that intent. Explicit time-series may be volatile or persistent, and the distinctive feature of this type is that they can be shared among multiple triggers. The reason for that is the independence of such time-series; i.e. they are not trigger-dependent like their implicit counterparts, and are external to triggers. Hence, multiple triggers may access the same explicit time-series.

The following is a description of the commands that may be used to create, manipulate, and delete explicit time-series:

- Creation command for a single time-series:

create time series *time_series_name*
[as *(attr(1), … attr(n))*]
from *data_source_name*
on *default_attr*
[when *qualfying condition*]
[timestamp *timestamp_attr*]
[with ([polling = interval], [storage = volatile], [window = interval], [status = private])]

In the previous grammar, the “as” clause defines the set of attributes a time-series maintains, if absent, then all attributes are assigned default values. The “default” clause specifies the default attribute to be assumed when the time-series’ name appears alone without an extension; i.e. when the attribute on which a continuous aggregate is to be applied is not specified. The “timestamp” clause specifies the timestamp field on which the time-series has to index the tuples or data. The timestamp field may be inherent in the original relation, or added by the corresponding data source. For example, in the function decrease(IBM, “10%”, 1 month), if IBM is a multiple-attribute data source, and the default attribute was set as the “closing_price”, then the system automatically maintains a time-series on IBM.closing_price. The “time” clause specifies the timestamp attribute of the row if available, if not, the temporal system will automatically time stamp any received version of the row. Other clauses have the same semantics as the ones used in other commands. Notice that in this command the created time-series maintains a history of any tuple that matches the when clause condition. Unless a primary key value is specified in the where clause, the time-series may contain histories of multiple tuples. An example where this option comes in handy could be as follows:

"A bank manager desires to monitor all the disbursed money from one of their ATM machines, in that case, the time series has to track the "amount disbursed" column of each customer that uses the ATM machine."

Here each customer will have a distinct record, but the application requires tracking a certain attribute of multiple rows. The following command will force the system to track histories per row, in other words, each distinct tuple or object will have an independent time-series to maintain its (or part of its) history.

- Creation command for rolling up a table into a collection of time-series (group by):

```
create time series collection collection_name
[ as (attr(0),  attr(1),  …  attr(n)) ]
from table_name
[on default_attr  ]
[when qualifying condition ]
group by (primary key attribute(s))
[ timestamp  timestamp_attr ]
[with  ( [ polling = interval ], [storage =  volatile], [window = interval ], [ status = private ] )]
```

In the second collection creation command, the primary key of the table must be transformed to a character string, so that it can be used to identify the time-series corresponding to each row/s of the table. The name of the time-series tracking the history of a row will consist of the name of the collection plus the primary key character representation. For example, if the collection name is stocks, and the primary key of the corresponding table is StockID, then the name of the time-series which will track a stock with StockID = "1220" will be "sotcks_1220". The "group by" clause will allow the system to group the updates according to the given primary attribute(s) in separate time-series.

- The defaults are:

  - no polling, i.e. on-line update (time-series has data inserted only when a relevant update is received)

  - storage = persistent

  - window = • (this may consume a lot of memory if the data size is large)

  - status = shared

- The manipulation commands:

  set polling = new_interval for time series time_series_name

  set status = persistent for time series time_series_name

- The deletion command:

  drop time series time_series_name

Example(1): Assume that a broker desires to maintain the history of IBM's hourly price, where each tuple in the stocks table has a time stamp attribute called "time", and the attributed to be used by default is called "closing". Here if the time-series name is the primary attribute (which is true here), then the time-series will maintain the history of the specified attributes (in the "contains" clause) for the specific tuples. If the stock name is not the primary key, it is very possible to find histories of multiple tuples in the same time-series.

```
create time series IBM_hourly_price
from stocks
contains (price, hi, low, closing, time)
index on time
default closing
when stocks.name = "IBM"
with (polling = "1 hour", window = "1 month")
```

Example(2): In this example, the intent is to transform a projection of the stocks table into a collection of time-series, where each corresponds to a single row in

the table, assuming that the rows are unique. The selection predicate will create a time-series for only those rows that have a closing price over $125.

```
create time series collection closing_TS
from stocks
contains (price, hi, low, closing, time)
index on time
default price
when stocks.closing > 125
group by stock_ID
with (polling = 1 day, window = 1 month)
```

The time-series in the previous example is a shared, persistent time-series that keeps a month's worth of hourly price history for the IBM stock. To delete the former time-series users can issue the following command:

```
drop time series IBM
```

If the time-series is explicitly created with a single-table definition, then all updates that filter through the corresponding selection predicate are immediately applied to the time-series. On the other hand, if the time-series is explicitly created with a multi-table definition, then it is treated exactly like a trigger in the sense that the system constructs a special discrimination network (filter network) for it. Unlike regular discrimination networks, filter networks only apply qualified updates to corresponding time-series rather than cause a trigger to fire. If an update tuple filters down into the network's P-node, it is eventually applied to the time-series, and no further actions are taken. This allows time-series to act as repositories of temporally materialized views. Sharing explicit time-series is controlled by semaphores to protect critical sections. Furthermore, this type of time-series has a higher priority when a time-series name is encountered in a function's argument list. This means that the time-series catalog is searched for an explicit time-series with the given name first, and if the search fails, then an implicit time-series is created.

TriggerMan's temporal system maintains a name-based time-series catalog that keeps track of all shared time-series. A shared time-series maintains the largest history window defined by any trigger utilizing the time-series. If a trigger applies a method on a time-series with a window larger than its existing one, the system automatically extends the time-series window to cover the required history window. As for recovery, Volatile time-series are reset (previous history is lost) each time the system shuts down, or fails.

Time-series may be defined on views, in which case, the view definition is incorporated in the definition of the corresponding data source. To avoid confusion and reduce complexity, the data source definition command allows users to specify their view definition in a uniform manner, and TriggerMan's server translates them into equivalent view definitions according to the source database's format.

4.6 User-Defined Time-Series

IDS/UDO provides a time-series DataBlade, hence, It is possible to have an attribute that is of "time-series" type. In this situation, if an attribute of "time-series" type is used in a temporal function, then the system must be able to access that time-series in order to apply the function correctly. Data sources have the ability to provide the system with user-defined time-series (attributes of "time-series" type) when needed. Once the system recognizes an attribute as a user-defined time-series, it sends a request to the corresponding data source in order to gain access to the time-series. On the data source's side, the update descriptors that include attributes of "time-series" type will have corresponding references to the time-series objects. The data source may choose to cache the time-series pointers in a special table or query the source on demand. Caching time-

series may reduce the cost depending on the time-series' size, access cost, and data availability. Timer-driven triggers processing may require accessing user-defined time-series regardless of updates. Therefore, whenever the system sends a request, the data source immediately supplies it with the required time-series pointer if it is cashed. Otherwise, the data source forms a query using the respective trigger's condition, then submits it to its database in order to get the time-series reference. It is obvious from the former discussion that user-defined time-series involve added complexity on behalf of the data source. Thus, the data source logic should be smart enough to handle such complex operations efficiently and robustly.

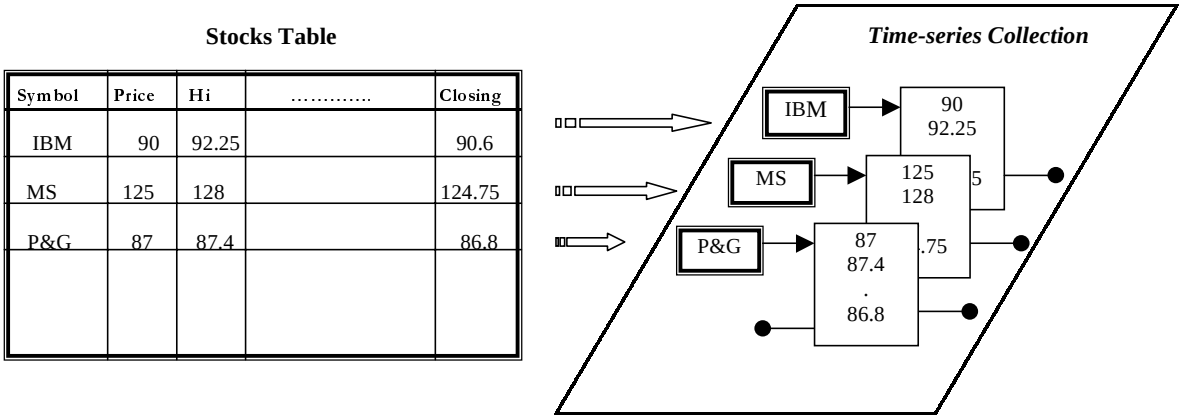

Figure 4-1: Rolling-up a table into a time-series collection

4.7 Time-Series Loading From Data Sources (Priming)

The system can load from a data source any time-series used by a trigger defined on it. As shown in figure 4-2, a data source can unfold a table in a database into a time-series collection at boot time or at trigger creation time. This facilitates the definition of rules on a table with multiple histories; i.e. the history of each qualified row in the table is

managed by an independent time-series. If the data is in the local database, then the time-series is loaded internally with minimal effort. Conversely, if the data is stored in a foreign database external to the system, then the data source has to issue proper queries to retrieve the needed data into the system's local storage. Depending on the size of the data transferred, the later situation could be substantially expensive due to communication overheads.

4.8 Sharing Time-Series

In most practical applications, many triggers may apply continuous aggregate functions on the same time-series as part of their conditions. In such situations, sharing time-series would be most appropriate whenever possible. For example, if a financial firm has a hundred employees, and most of them are interested in IBM's stock value from a different perspective, then each of them would create an IBM trigger that serves his/her purpose. It makes sense in this case to have all the interested brokers' IBM triggers apply their related aggregates or functions the same time-series in order to save storage space.

Since implicit time-series are trigger-dependent (they are created only when a trigger is, and their contents depend on their selection predicate) they cannot be shared due to the associated implementation complexity. Conversely, explicit time-series are totally independent and autonomous object, and hence, it may be shared among multiple triggers. The only restriction here is the time-series maximum window, which is an attribute of the time-series. If the trigger requires operating on a larger window than the time-series' existing one, then the user has either to alter the time-series to fit the trigger's requirements, or create a new one to suit his/her trigger definition.

Sharing time-series has a negative repercussion; that is, simultaneous access to the time-series by a large number of triggers might slightly attenuate the performance due to locking conflicts. Hence, users are advised to avoid using common explicit time-series for performance-sensitive applications.

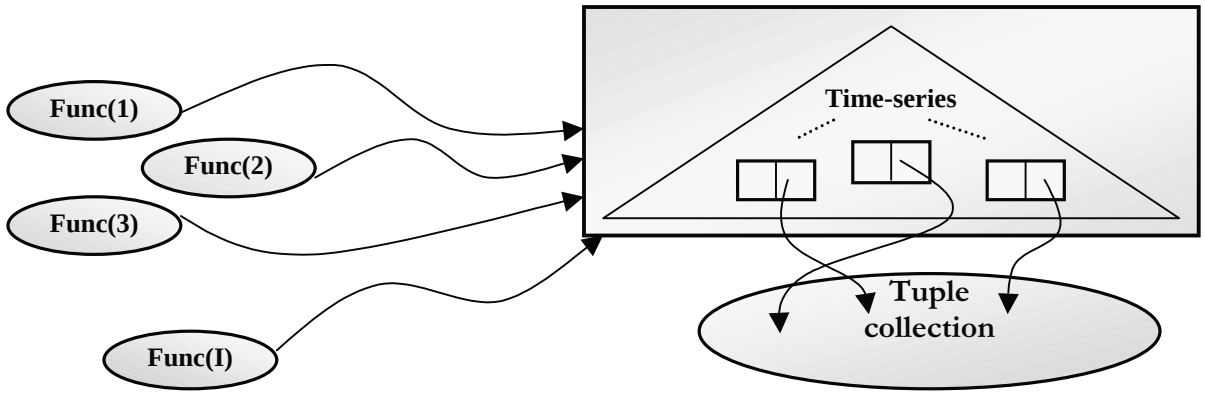

Figure 4-1: Sharing an explicit time-series amongst multiple continuous aggregates

4.9 Time-Series Recovery

In database systems, recovery is an extremely important issue. Reliability and fault-tolerance of the system depend greatly on the recovery technique(s) the system deploys in case of a crash or a failure. Transactions are the basic recovery unit in database systems. Thus, a running transaction should be guaranteed to safely commit or abort if the system crashes or fails during its lifetime. Logging is the most common recovery approach used in database systems. In case of temporal data, the problem is different because in many temporal applications the data is unpredictable and is usually supplied by external agents or modules. If the data is on-line and not stored in the source, then unless a retransmit or a recollect protocol is used, it is hard to get hold of the data

that arrives during a system failure or crash. Conversely, if the source stores the temporal data in a temporally queriable format before it is delivered to TriggerMan, then recalling the data lost during the down period is possible. For the sake of implementation simplicity, in the first stage of TriggerMan project, the system does not employ any recovery schemes. Therefore, system crash and failure events will show as no-update periods in the present collection of time-series reflecting the loss of data updates if there were any.

If the time-series were volatile, then their data would be completely lost if the system crashes or fails for any reason. In that case, the system rebuilds the time-series from scratch starting at the time of recovery. On the other hand, if the time-series were persistent, then only the data updates that arrive to the system during its down period are lost. Most of the realistic temporal applications deal with large time granularities such as hours, days, months, etc. Furthermore, system crashes are rare and their durations in most cases are relatively short and are measured in minutes or at most hours. Hence, many applications would be able to tolerate such minor system crashes or failures. Currently, TriggerMan has no ability to retrieve the temporal data lost during a crash event or system failure. Depending on the application, such a gap in a time-series can be filled using different methods such as interpolation, data recall (if the data is stored), etc. Another alternative would be the queued transactions-like technique [Bern90]. According to this approach, the remote data source locally collects the updates in a queue, and the system dequeues updates directly from there. This way, even if the system crashes, the updates will not be lost because they are being collected at a remote location.

In a more advanced version of the system, a recovery technique may be designed and implemented if it proves necessary.

4.10 Correction of Time-Series Data

Updating a time-series means the addition of a new value(s) at a certain point in time to the time-series data. Inversely, correcting a time series means replacing a specific value(s) in the time-series because it is either invalid or incorrect. The distinction between correction and update of temporal data was addressed in the temporal databases literature [Snod86, Abb87, Ros91]. Some studies paid close attention to the temporal data correction issue [Gal93]. Some temporal databases provide several temporal data correction commands as part of their languages to enhance their temporal querying capabilities. TriggerMan is a rule system and does not provide temporal query capabilities at this stage, hence, only simple temporal data corrections are needed. Accordingly, TriggerMan provides a simple command for replacing a faulty value with another one or completely removing it from the history as if it has never occurred. The following command is used to achieve the former task:

update time-series - log_time_series *time-series-name* at *timestamp* with value = (new value | NULL)

The following pseudo code illustrates the mechanics of the previous command and shows the details of its implementation:

```
If ( timestamp != NULL )
{
        value = get_value(time-series, timestamp);
        if (value != NULL)
                update_time_series(time_series, timestamp, newValue);
        else
                insert_time_series(time_series, timeStamp, newValue);
}
else
{
```

```
        if ( value != NULL)
                update_time_series(time_series, get_last_timestamp(time_series), newValue);
        else
                delete_time_series(time_series, get_last_timestamp(time_series));
}
```

The “log time-series” option creates a log time-series (if not available) for recording the fixes that have been applied to the corresponding time-series. After each time-series update, the system checks if it has a log time-series, if it finds one, it records the old and the new values of the update that has taken place. Such a log time-series is useful to track problems or a mishaps. This is helpful especially in decision support analysis.

Correction operations are semantically different from regular database operations. The effects of the correction operation are limited to directly replacing or deleting an instance of the time-series. On the other hand, database operations have further effects in terms of rule processing. Namely, an insert, an update, or a delete operation may cause a rule or a set of rules to fire whenever they occur. Correcting a history maintained by a time-series implies a previous error or a change in the decision that produced the respective data of a time-series instance depending on the application. Therefore, it is our belief that the corrected value(s) belong to a past timestamp and it should be understood in the context of either the following two cases:

- The value of the corrected instance should have been: the corrected value(s) or
- The value should not have occurred.

Consequently, such a correction should not interact with any rules defined in the system because firing rules as a result of obsolete updates does not always make sense for most practical applications.

CHAPTER 5
TIMER MODEL

Presently, many commercial DBMSs provide sophisticated triggering capabilities. Examples of such systems include *Oracle, Rdb, Ingres, Informix, InterBase,* and *Sybase*. As for temporal trigger capabilities, only a few products provide tools for that (RapidBase). Since temporal applications deal with different aspects of time, it is essential for those applications to have the ability of monitoring the time line. Furthermore, temporal applications should be able to detect accurately specific temporal events as soon as they occur, and invoke particular actions accordingly. Therefore, one important feature a temporal trigger system should possess would be a timer system. Timers are essential for time-sensitive triggers and applications; furthermore, they can be used to simulate batch job schedulers. Hence, one main feature of TriggerMan's temporal rule processor is the timer system, which is a complete facility that can be utilized to serve diverse timing purposes. To the best of our knowledge, none of the mainstream commercial products support a full-fledged timer tool that is as elaborate as the one we propose here for TriggerMan's.

This chapter presents the conceptual design of the TriggerMan's timer system along with selected implementation details. Towards the end of the chapter, a few examples are given to illustrate the importance and usefulness of timers.

5.1 Timestamp Formats

The timestamp is an essential element of our temporal trigger system. There are several formats for representing the date/time as a timestamp. This section presents a clear definition of the different timestamp formats, and confers any related assumptions. Timestamps may be relative, absolute, or periodic-absolute. Relative timestamps simply specify offsets from a reference time point, and define a valid time window that is represented by the interval [(*reference time point – offset*), *reference time point*]. Our system implicitly employs a past temporal logic, where the relative duration extends for the *current time* back into the past. An example of a relative timestamp would be saying "for one month" which is syntactically represented as "1 month" in our language. The former timestamp instructs the temporal system to consider only the respective data with timestamps falling within the interval [("now" – "1 month"), "now"]. Absolute timestamps define a specific time point that occurs only once. For example: # 12/1/1997 10:30:00 AM # is an absolute timestamp which can be translated into the following date/time specification: "1st of December 1997 at 10:30 AM." Periodic absolute timestamps contain one or more wild cards to indicate specific periodic timestamps. To achieve expressiveness, multiple wild cards may be used in a timestamp expression to specify an absolute-periodic timestamp. As will be described soon, the system applies a wild card substitution method to an absolute-periodic timestamp in order to get the next proper absolute timestamp if applicable.

The rest of this section focuses on using wild cards to implicitly define periodic-absolute timestamps. For example, using our language, a simple periodic timestamp to represent the following periodic timestamp: " each minute of 1 PM on 1st of January

1998” would be written as: # 1/1/1997 1:*:00 PM #. On the other hand, a timestamp that represents the following date/time description: “the 1st of each month in 1998 at the last minute of every hour” would require multiple wild cards in order to capture the right semantics. Hence, the former example would be written as: # */1/1998 *:59:00 PM #. When the timer system receives a request to schedule a timer for a periodic-absolute timestamp, it first replaces the wild cards with the temporally nearest equivalent values. For example, in the event specification # 1/1/1997 1:*:00 PM #, if the current time has not reached # 1/1/1997 1:00:00 PM # yet, then the system will replace the “*” with the smallest future value of minutes which is 00 to produce the next wakeup timestamp for the corresponding timer. Otherwise, if the time is already within the specified range, for example, if the current time is #1/1/1997 1:30:00 PM #, then the system will replace the wild card with the smallest future minute value that is 31 in this case. Once the timer system substitutes the wild characters in the date/time string, the resulting value (which is #1/1/1997 1:31:00 PM # in the former example) is properly inserted into the wakeup list as the wakeup time for the respective timer. Once the timer system wakes up such a timer, it refers back to it absolute-periodic timestamp string and repeats the substitution process. Each time the timer system processes a timestamp string (wild card substitution) , it checks to see if the resulting actual timestamp has become stale (it is earlier than the current timestamp). If the resulting wakeup timestamp turns to be stale, the system simply does not reschedule that wakeup time and deletes the corresponding timer’s instance. For example, in the string # 1/1/1997 1:*:00 PM #, if the current time is # 1/1/1997 1:59:00 PM #, then the smallest future value of minutes when the time becomes #1/1/1997 2:00:00 PM # is 00. In this case, the next wakeup time becomes # 1/1/1997

1:00:00 PM # which is stale with respect to the current time. Hence, the timer system will not reschedule this wakeup time and it will delete the corresponding timer's instance.

The same substitution procedure is followed for multiple wild cards, for example, suppose that there is a wild card for the months and one for the hours in some timestamp string. In such a situation, the system substitutes the hours' wild card with the smallest future value of hours and the months' wild cards with the smallest future value of months. Afterwards, it checks the validity of the resulting wakeup timestamp, and schedules it if it is not stale.

It should be noted that if the hours, minutes, or seconds fields are omitted, then they are assumed to be "00" by default. Therefore, omission of a field is semantically different from having a wild card in that field.

5.2 The Timer System Architecture

TriggerMan employs a rather simple timer model to serve the purpose of detecting the different types of temporal events. Figure 5-1 reveals the basic architecture of the timer system. The main components of the system and their responsibilities can be listed as follows:

- *The timer manager*: Responsible for checking the wakeup list each clock tick and running the action of the awakened timer(s). It also accepts timing requests from timer objects and properly inserts them into the wakeup list,
- *The wakeup instance*: a structure that holds a real time value at which the corresponding timer needs to wakeup,
- *The wakeup list*: this is simply an ordered queue that maintains a list of timer wakeup

instances, and

- *The timer object*: This is a structure created and initialized whenever a trigger that requires a timing function is created.

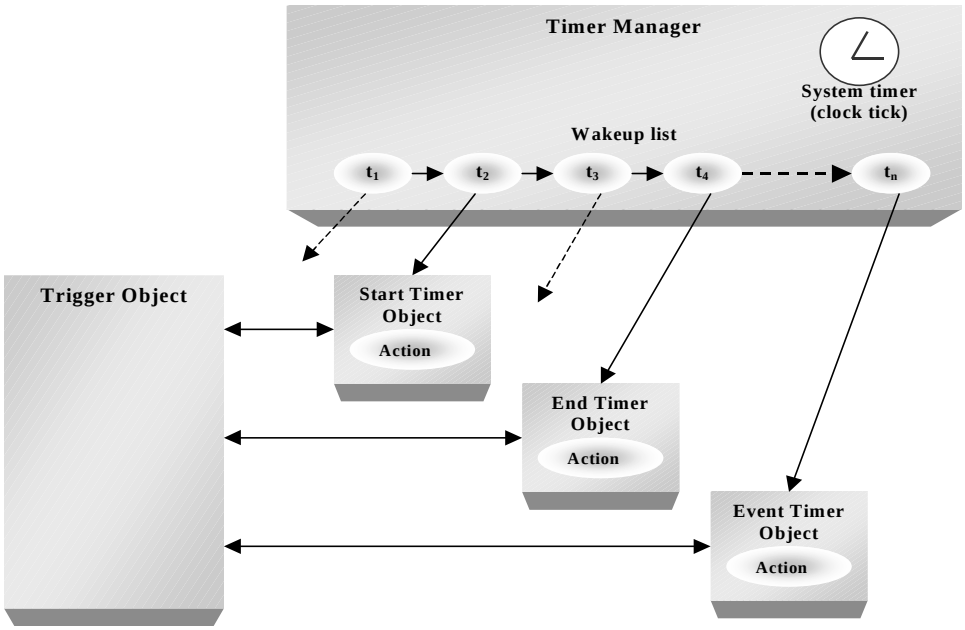

Figure 5-1: Timer system architecture

TriggerMan identifies three types of timers that may be used within a trigger definition as shown in Figure 5-2. Event timers are the most important of all because they alter the semantics of the trigger condition when used. Any trigger that uses temporal events represented by a timer specification is categorized as a timer-driven trigger; otherwise, it would be a regular event-driven trigger. The former two categories of triggers are discussed in detail in a subsequent section. The other two timer types are the start and end timers. As the name indicates, start timers are used to specify the time for automatic trigger activation. Conversely, end timers are used to specify a time for

automatic trigger deactivation. Although our language provides commands for activation and deactivation of triggers, it requires the user to issue the command when needed, which might be undesirable in many cases. In such situations, automatic activation and deactivation come in handy.

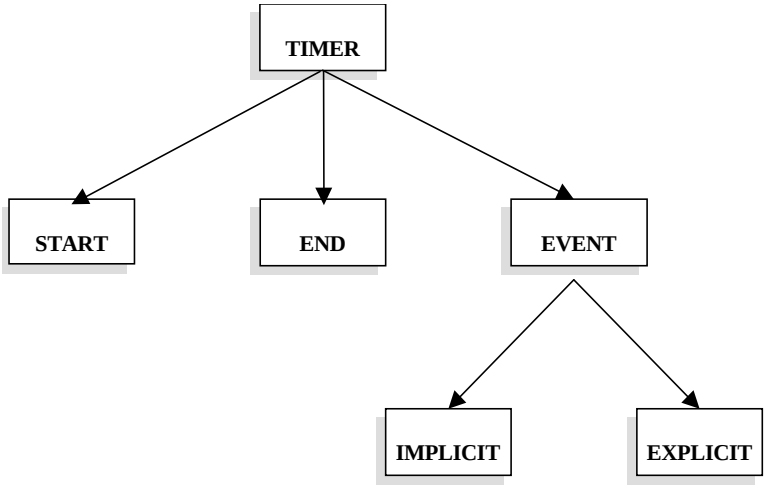

Figure 5-2: Timer categories

5.3 Timer-Driven vs. Event-Driven Triggers

We classify all triggers with respect to their condition evaluation causality into two types: event-driven triggers, and timer-driven triggers. Event-driven triggers require a database event (insert, delete, update, and read) to cause its condition to be tested. The trigger may be temporal or not. Timer-driven triggers, on the other hand, are different in the sense that they only respond to timer wake up calls, at which point they test their conditions. This categorization simplifies the trigger logic, and provides a clean model for handling different kinds of rules.

Event-driven triggers depend on update descriptors delivered to the TriggerMan system by what is called a data source. A data source is simply an interface layer between TriggerMan's server and an external DBMS. When a data source receives an update descriptor, which might contain an insert, delete, or update, in addition to the data itself, it directs it to an event queue. All the triggers that have conditions involving the data of the received event will have their condition tested. If the condition evaluates to true, then the action of the corresponding trigger will be executed as soon as possible (we say the trigger fires in that case).

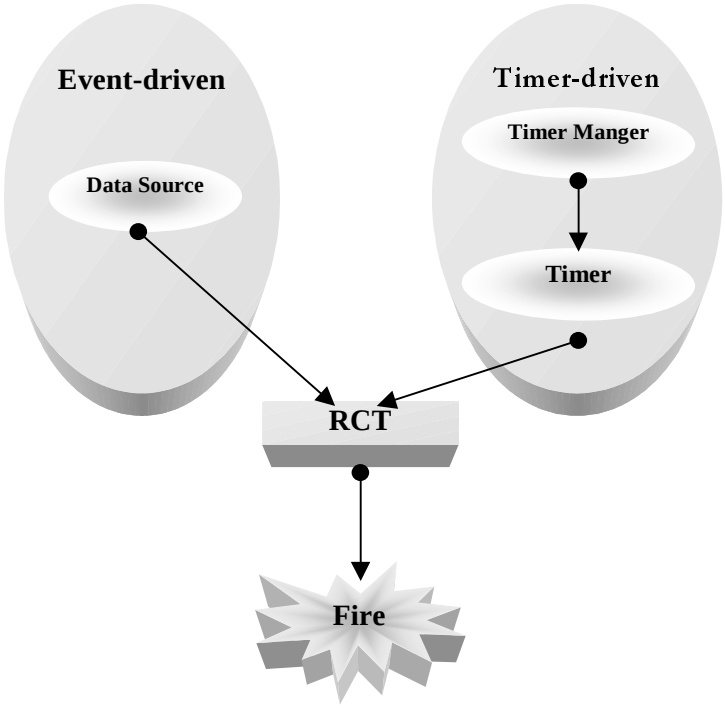

Figure 5-1: Event-driven vs. timer-driven triggers

For example, it is imperative to define a timer on temporal operators like "StayInRange" or "Holds". For these two operators, the value of a time-series might stay in range or a condition might continuously hold respectively for some time even without

applying any updates to the respective time-series. In contrast, it would be meaningless to define a timer on event-dependent operator such as “increase”, because there is no way for the condition of such function to change unless an update to the database takes place. Thus, the use of timers should be restricted to useful applications only, otherwise, it may cause useless condition testing. Only built-in timer-driven temporal operators inherently have timers. Since we have no control over user-defined operators, it is up to the users to make any temporal function be timer-driven, even if it is meaningless. To elaborate further, if a trigger has an “increase” operator in its temporal condition, then defining this trigger as timer-driven makes no sense. That is true because such an operator is event-dependent, in other words, its value might change only when an insert, an update, or a delete takes place. Thus, even if the timer goes off every certain period, the trigger might fire only when the system captures a database event, therefore, having the trigger be timer-driven would be useless is such a case. Users should be smart about using timers with their own custom-made temporal operators. The syntax of timer-driven triggers would be as follows depending on the timing mode:

- Default timing:

```
Create trigger trigger_name
on timer
.
.
having  timer_driven_temporal_function( )
.
.
```

- User defined timing:

```
Create trigger trigger_name
on timer ( timer_duration or  #day/time# )
. . .
having timer_driven_temporal_function( )
.
```

As described earlier, when the system encounters a timer-driven trigger, it immediately creates a *timer* for it, and initializes the timer with the given *duration* in case of a user-defined timing. If no timer-duration was specified (default timing), the timer will be initialized with the function's time window duration by default. Whenever the timer goes off, the system evaluates the temporal function as part of the temporal condition, executes the trigger's action if the result is true, and restarts the timer again. A simple example to illustrate the user defined timer concept would be:

```
create trigger …
on timer  1 day
.
.
having stayInRange(X, "10", "20", "1 mo")
.
```

The former temporal condition means: check every *day* whether the time-series value has stayed between 10 and 20 in the past month. Thereupon, execute the trigger's action if the result is true. Without a timer, this condition would be checked only when a database update occurs. In view of this, if a value stays in range for a whole year, and no updates to the time-series take place throughout that period, the condition never gets checked. Consequently, the trigger would never fire, leading to an undesired behavior. The following example illustrates the significance of timer-driven triggers:

```
create trigger WeekStock  from  SP500
on timer 1 day
 having  decrease(SP500.closing, " 10%", "1 wk") and
         ( maxIncrease(SP500.closing, "1 wk") < 2% )
group by SP500.StockID
do   Raise event SellStock  SP500.StockID
```

This is actually a stock monitor, which periodically (every 1 day) checks the weekly behavior of the stocks in a particular portfolio, and takes a specific action when the condition evaluates to true. Here, the trigger disregards any database events, and

depends solely on the defined timer. If the "on" clause is omitted, then the semantics become entirely different. In that case, the trigger would be a hybrid trigger, and any relevant updates might set the trigger off. Thus, users should have a clear distinction between the former trigger types, and correctly write the trigger according to their application requirements.

From an implementation perspective, whenever a user creates a trigger, the system may create a timer object as part of the corresponding trigger's state or environment. The cases in which the system creates a timer are

- The trigger is timer-driven; which means that the trigger's event is timer-based,
- The start-time clause is specified, or
- The end-time clause is specified.

Timers can be implemented in various ways depending on the underlying system. High-level details of the timer management system are described in the following section.

5.4 Timer Implementation Issues

In the stand-alone implementation option of TriggerMan, the timer system may be implemented as a multithreaded module. In this case, a scheduler thread and an action thread are required to accept requests and monitor the wakeup list respectively. Since multiple timers may go off within a short interval, an action thread may be spawned for each awakened timer. This enables the timer actions to run in parallel, possibly enhancing the performance. A shared queue could be used for submitting timing requests by the command server, and scheduling them by the timer scheduler.

In the IDS/UDO-coupled implementation option, the same timer system may be used. In this case, programmers have to be vigilant and extremely careful when coding the former timer system implementation scheme. Such code has to be thread-safe and system compliant in order to be compatible with Informix operation and to avoid interference problems. Alternatively, The TriggerMan server accomplishes the timing functionality by receiving an external clock tick signal. The same data structures used in the original scheme may be utilized. In this setup, when TriggerMan server receives a clock tick, it checks the wakeup queue and compares the timers' real time to the current time. If a timer qualifies, meaning that its wakeup time is less than the current time, then the system immediately executes its action. In the stand-alone architecture of TriggerMan, the timer system was implemented as two separate threads for timer request queuing and timer wakeup time scheduling respectively. In the Informix-coupled architecture, we avoid creating threads for the timer system by having an external clock tick be delivered to the system each second. At each time tick, the timer system retrieves the current timestamp, compares it to the scheduled wakeup timestamps, and if any matches, the system executes the corresponding timer's action. Since our system was not designed for real-time applications, we believe that one-second granularity is good enough for most real life applications. An external source can possibly provide our timer system with the required clock tick. On the other hand, if the TriggerMan driver invokes the system every one second, then invoking the timer system at the same time can be substitute the external clock tick. This way, the timer system will always get the current timestamp and check its wakeup list as soon as its invoked implicitly realizing that one second has already passed since its previous invocation.

5.5 Timer Application and Utilization

There are numerous applications where timers play a significant role. This section lists some examples to illustrate the use of different timers. Furthermore, the following examples provide some leads on how to use timers efficiently in order to satisfy application requirements.

- Batch processing:

"Every day at 8:00 am, run a daily batch job"

```
create trigger batchProc
on timer # 8:00 am #
do exec daily_batch_job( )
```

- Trigger activation control:

"During Christmas season of every year, store all the sales in a special table called christmas_sales_table"

```
create trigger christmas_sales
from sales
on insert
start on # 12/1/* #
end on # 12/25/* #
do insert christmas_sales_table values (:NEW)
```

- Timer-driven testing:

"On the 1st of each month, check the monthly sales of each representative in the sales department, and report to the supervisor any one who had sold less than 75% of their monthly quota"

```
create trigger low_sales
from daily_sales, representatives
on timer # */ 1 / * #
where daily_sales.rep_id = representatives.id
having avg(daily_sales.amount, "1 month") < .75 * represetatives.quota
group by daily_sales.repid, daily_sales.supervisor, representatives.quota
do exec send_message(daily_sales.supervisor, daily_sales.repid, "below quota")
```

Another possible application of timers would be batching high frequency updates to materialized derived data [Adel97]. Applying single-update to materialized derived data or views requires recalculating the corresponding base data. If the update frequency is high, the recalculation overhead can swamp the system's CPU and degrade its performance. A good strategy to reduce the recalculation overhead when the update frequency is high is to batch updates periodically and apply all the updates collected in the batch at once. Such a strategy is ideal when the single-update changes are small or when the data change rate is graceful. This relieves the system from the burden of single-update recalculations and helps to enhance its performance. To implement update batching in TriggerMan, the timer system should be used to time batching cycles. Furthermore, the system needs some data structure to collect the updates in order. Time-series are the perfect choice to collect updates in temporal order. The server logic has to be slightly modified to accommodate update batching. Instead of testing an update against a respective trigger's condition, the update is stored in an update-record time-series. When the timer goes off, all the updates contained in the update-record time-series are tested against the corresponding trigger's condition. Once an update matches the condition, the proper action is executed.

The update-batching scheme is not of great concern because the present TriggerMan server does not provide any facilities for derived data maintenance. If the system is to maintain derived data in the future, and high frequency updates are expected, then updates batching becomes an issue of concern.

The update-record is kept on a per-trigger basis. This means that the system creates an update-record for each trigger that involves derived data and uses the batching

scheme. When the system receives an update, a copy of its update-record (or update descriptor) is inserted into any update-record time-series that is concerned with the received update. Thus, when a timer goes off in an update-batching trigger, the system looks up its update-record time-series and applies all the recorded updates if they match the condition. After all the updates in an update-record time-series have been processed, the system clears the time-series and resets the timer to start a new batching cycle. If the batching cycle is set to a long duration and the update frequency is high, then the number of updates can be huge. Consequently, a substantial memory space can be exhausted by update-record time-series. In order to achieve optimal performance, it is best to empirically calculate the suitable batching cycle for each individual application, which depends mainly on the updated data size and frequency.

CHAPTER 6
PARALLEL TEMPORAL TRIGGER PROCESSING

The issues of parallel temporal trigger processing are not much different from those of regular parallel trigger processing. Such issues are explored and discussed in many studies [Ish91, Stol91, Wolf91, Wolf93, Alf96]. Here, the discussion will be focused on parallel issues with particular attention to temporal function evaluation. The major difference in temporal triggers is the temporal condition, which, in most cases, involves the time dimension. Temporal conditions simply constitute an expression that consists of temporal functions connected by Boolean operators. Thus, applying each of the temporal functions to the corresponding time-series would be the major parallelism issue within the scope of this work.

TriggerMan implicitly runs on a virtual processor (VP) architectural platform. Consequently, the parallelism granularity (unit) in the rest of the chapter will be the VP. Ideally, one VP is spawned on each processor in the system, however, multiple VPs may share one processor if the resources are tight. In IDS/UDO, VPs are dynamically allocated to physical processors, hence, a VP may move from one processor to another during its lifetime.

6.1 Parallel Testing of Complex Temporal Conditions Via Remote Function Evaluation

To take advantage of parallel processing, multiple VPs may evaluate a single temporal condition simultaneously. If the time-series are distributed among all VPs in the

system, then multiple time-series temporal conditions may be tested in parallel. Time-series may be hash or round robin partitioned. Hashing may be based on a name or an id in a way that guarantees load balancing, such that the average number of time-series is roughly equal on all VPs. The grouping attributes in a time-series collection may be used to hash partition the collection's individual time-series across the available VPs.

When a trigger's temporal condition is to be tested, all functions and aggregates have to be applied to their respective time-series. A VP will have no problem applying a function on a local time-series because it will have direct access to the time-series. If the time-series is located on a different processor then the VP has to send an application message to the VP that has a local access to the time-series. This is called the *remote function evaluation*. An application request message includes the time-series name, and the function that needs to be applied to it along with its arguments. Whenever a VP receives an application request message from another, it extracts the time-series name, the function and its arguments, applies the function to the time-series, and finally replies to the VP that sent the application request.

This way, the rule condition tester of one TriggerMan VP can test a complex temporal condition in parallel rather than doing all the work locally in a sequential fashion. This scheme incurs slight communication overhead. If a trigger's temporal condition contains one or more time-series applications, then the trigger is guaranteed to fall on a VP which locally maintains at least one of the condition's time-series. This guarantee will help reduce the communication overhead. Practically, not all triggers have multiple time-series applications in their conditions; therefore, many of the simple triggers will be tested locally and no communications will be required. The next two

sections discuss details of remote function evaluation under the SMP and the shared-nothing architectures.

6.1.1 Distributed Processing of Time-Series on a SMP Architecture

In shared-memory architectures, VPs utilize the shared memory for communication. The processes can exchange all kinds of data by having the provider post the data in a specific location in memory where others can grab it and use it according to their needs. In temporal trigger processing, it is essential to utilize the system resources by having processes cooperate to handle large jobs in parallel. Thus, when a temporal condition is complex (contains function applications to multiple time-series), it would be intuitive to apply the functions in the temporal condition to the time-series in parallel. This is feasible if the time-series are partitioned across the processors in the system. A distributed scheme can be used to achieve parallel time-series application. According to this scheme, each VP maintains two queues, a *request queue*, and *a reply queue*. The request queue of a VP is used by other VPs to insert their requests for time-series function application. On the other hand, the reply queue is used by VPs to deposit the results of requested time-series function applications by the local VP. Hence, when a VP needs a time-series function application, it has to know which VP maintains the time-series, which is called the target VP. Thereafter, the requesting VP enqueues a data structure that contains the requesting VP's id, time-series name, function name, and the function's argument list in the target VP's request queue. The target VP periodically (when it is free) dequeues an entrée off its request queue, extracts the function application attributes and applies the function. When the result is returned, the

target VP queues a data structure that contains its id, the function's id, and most importantly the result in the requesting VP's reply queue. Figure 6-1 illustrates remote function application in a shared-memory environment.

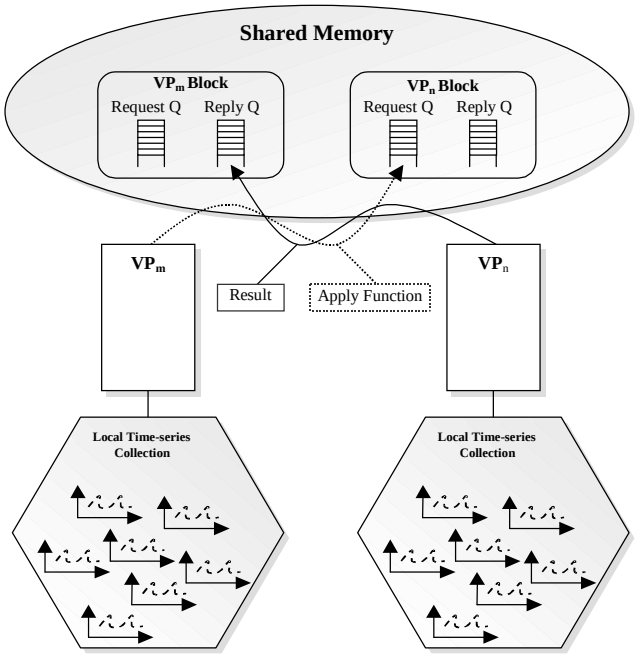

Figure 6-1: Remote function evaluation in a SMP environment

6.1.2 Distributed Processing of Time-Series on a Shared-nothing Architecture

In a shared-nothing architecture, there is no shared memory that can be utilized for communication; rather, message passing is the only means to cooperate as shown in Figure 6-2. In this case, if a VP needs to apply a function to a remote time-series located on another VP, it has to send an application message to the respective VP. In order to achieve a correct operation, there needs to be a protocol where a VP can directly send an application message to the VP that locally stores the respective time-series rather than broadcast the message. One possible solution is to have a replicated time-series catalog

which maintains all the time-series in the system along with their locations (VPs). This would work fine, however, it is inefficient due to the redundancy, and the overhead associated with updating the catalog for any reason. A centralized version of the previous scheme is a time-series-location server scheme, where one of the VPs would maintain a time-series catalog that is similar to the one used in the distributed scheme. Again, this solution suffers from all the deficiencies of centralized processing, and requires much higher message-passing rate. This, in turn, would increase the network traffic. An alternative solution would be to apply a hash-based scheme for partitioning the time-series across all the VPs and distribute the hash function across the VPs. In this scheme, no catalogs are needed, and the replicated information (the hash function) is static and requires no maintenance. The overhead incurred from applying the hash function to a time-series attribute is smaller compared to the overhead of searching the time-series catalog.

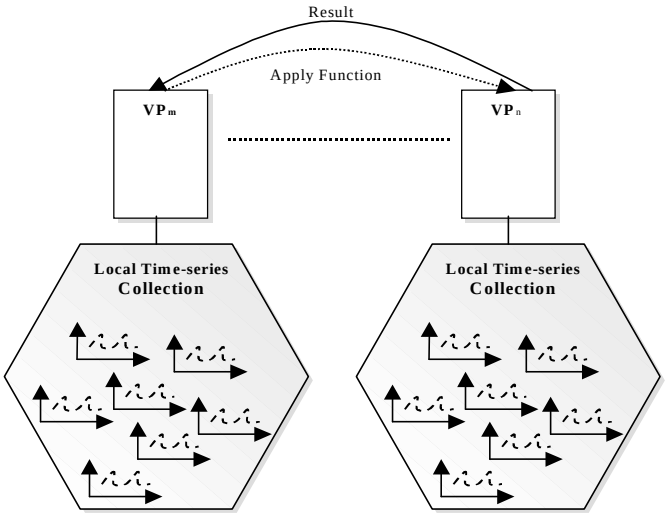

Figure 6-1: Remote function evaluation in a shared-nothing architecture

CHAPTER 7
LINGUISTIC COMPARISON BY EXAMPLE

Here we strengthen the motivation for our research, which produced the framework of this dissertation. Specifically, this chapter presents a linguistic comparison between our language and other available products and shows how we believe it proves superior. The main contributions in this area as discussed in the related work chapter were introduced by [Sist95a, Sist95b, Mot97]. Therefore, the comparative study considers the former two active temporal languages (TREPL and FTL) and weighs them on the scale against our active temporal language. Linguistic comparisons (especially computer languages) are not trivial and there are no standards or manifest measurements to serve this purpose. Hence, this comparative study relies primarily on practical and aesthetic criteria rather than theoretical measurements. Our temporal trigger language was designed in part to overcome the shortcomings of other such languages. Nevertheless, the reader should note that throughout this chapter, we try to look at the different languages from an unbiased (as much as possible) user's perspective. In that sense, even if a language has superior capabilities, if it is ambiguous, complex, or does not provide simple tools to utilize the available capabilities, then it will score low in our scale. Consequently, complexity, readability, user-friendliness, and flexibility define the collective focal point of this concise comparative study. The following are a few selected examples from different application fields that illustrate the superiority of TriggerMan's declarative language.

Example (1): "if the closing price of a stock rises by more than 20% in one day, to say, P, report its price on every day following that rise, as long as it remains above P."

═ In TREPL:

```
monitor update(stock_closing) as st_cl
StarRule:
        ( st_cl(X, X.new_price > 1.2 * X.old_price),
        * st_cl(Y, Y.ID = X.ID, Y.price >= X.new_price) )
→ write(Y.ID, Y.price)
```

In this language, users must have adequate programming skills to be able to identify the temporal variables involved in the temporal event. Thus, in the above example the event testing requires three values in order to check the 20% increase in the price to a value P, then test if the subsequent values stay above P. In this example, the three required values are the old price that is maintained by X.old_price, the newer price of the stock that is maintained by X.new_price, and the newest price that is maintained by Y.price. The conditions must be explicitly specified as in "X.new_price > 1.2* X.old_price" which is similar to writing a high-level programming language (3GL) code. Furthermore, these conditions must be contained within explicit update descriptors that capture the update information such as st_cl. These explicit variables and update descriptors contribute to the complexity of the rule definition and result in a poor readability. The star notation "*" indicates that the event must occur at least once. We use the same notation in our regular expression evaluator because it is widely used and understood.

✎ In FTL:

```
Condition:
[X ← stock_closing.price][t ← time]
eventually [ Y ← stock_closing.price && Y > 1.2 * X] [ t1 ← time && t1 > t]
```

```
eventually [Y > stock_closing.price && time > t1]
Action:
write(stock_closing.ID, stock_closing.price)
```

FTL is similar to TREPL in its need for explicit variable definition. In this example, the stock price has to be stored in the variable "X". Furthermore, the timestamp of the value stored in "X" has to be explicitly stored in a variable which is "t" in this example. For the subsequent values of the stock price, another variable "Y" is needed to capture the new value. The user has to explicitly keep track of the timestamp order using time variables such as "t" and "t1". This is even more complex and tedious than TREPL because the users is accountable for specifying conditions using temporal variable and specifying the order of the corresponding timestamps. The "eventually" operator denotes a possible time gap before the occurrence of the enclosed event.

✒ In TriggerMan:

```
create trigger 20%Raise1 from stock_closing

having  Reg_Expr_Eval( increase(stock_closing.price, " 20% ", " 1 day ")
                       → *(No_drop(stock_closing.price)) )
group by stock_closing.ID
do begin
      write(stock_closing.ID, stock_closing.price),
end
```

Here, neither temporal nor timestamp variables are needed. The temporal condition is expressed using temporal functions that take certain parameters and test for specific temporal events such as increase, decrease, and so on. In this example, a regular expression evaluator takes two events connected by a sequence operator (→ which equivalent to "then"). The first event is a 20% increase in the stock price to a value P, and the second one is a recursive (the "*" means it should occur once or more) no drop under P. It should be noted that the "noDrop" function used here is an extended function rather than a built-in one, which means that it is written and registered with the system by

a user. As described earlier, our temporal system implicitly maintains the history of the temporal variables in order for the temporal function to operate on those histories. It is obvious that the above condition is much closer to the English description of the temporal trigger. In addition, it relieves the user from going through the hassle of defining temporal variables. Therefore, The complexity of defining the trigger is reduced, at the same time; the readability of the trigger syntax is substantially enhanced. Overall, the language is more declarative and less procedural than TREPL and FTL.

Example (2): (Quantified Aggregates) "For every stock in a portfolio, derive its phases of monotonically increasing closing price that lasted at least 10 days."

⇒ In TREPL:

```
Monitor update(stock_closing) as st_cl
Good_Phase:
(* st_cl(X, X.ID = First(X.ID), X.price > Last(X.price)),
st_cl(Y, Y.ID = X.ID, Y.price < X.price, Count(X.ID) >= 10)
)
→ write(" Good phase for %s\n", X.ID)
```

This example requires a trigger to track the value of each stock in a portfolio for ten days. The grouping of stocks on ID is achieved using the First(x.ID) operator which locates the set of stock values that has the same given ID. The rest of the condition checks to see if there was a monotonic increase in a respective stock price for ten days. Since the constructs used here are the same as those used in the previous examples, there is no need for further discussion.

✎ In FTL:

```
Condition:
[st_cl ← stock_closing.ID][x ← st_cl.price][t ← time]
eventually [st_cl.price > x  &&  time >= t+10]
Action:
write(" Good phase for %s\n", stock_closing.ID)
```

The trigger for this example looks more compact than TREPL. However, the temporal variables such as “x” and “t” are still needed. The variable “x” maintains the first price of the stock, and its timestamp is recorded in “t”. All what is needed now is to make sure that all the price updates that follow the initialization are greater than “x” during a ten-day period. This is accomplished using the “eventually” operator in conjunction with the variables comparison as shown above. The noticeable problem here is that the time granularity is fixed and assumed to be in days. This firmly restricts the user and limits the language’s capabilities.

✒ In TriggerMan:

```
create trigger 10_day_mono_increase from stock_closing
having diff(stock_closing.price, “10 day”) > 0 and
        max_decrease(stoc_closing.price, “ 0 ”, “ 10 day ”)
group by stock_closing.ID
do raise event good_phase (stock_closing.ID)
```

For this example, the condition simply requires two functions as shown above. The first function would check for an increase during a ten-day period. The second condition guarantees that there were no decreases phases during the previous ten days. If the conjunctions of the two functions results evaluate to true, then, that indicates a monotonic increase during the past ten-day period, hence, the trigger fires. Here, the trigger needs to check the temporal condition for any stock in the portfolio, which requires grouping the updates on the primary key. The “group by” clause is used here in order to achieve grouping by ID. This is obviously more intuitive than manually trying to do the grouping as in the other languages. In addition, unlike the other languages, the time granularity is variable and user controlled.

Example (3): (Running Aggregates) “If the closing price of ‘IBM’ rises by more than 20% in one day, to a price, say P, report every new maximum in the following working days, provided that the closing price has remained above P.”

═ In TREPL:

```
monitor update(stock_closing) as st_cl
New_Max_So_Far:
( st_cl(X, X.ID = 'IBM', X.new_price > 1.2 * X.old_price),
  * st_cl(Y, Y.ID = 'IBM', Y.price >= X.new_price), Y.price = Max(Y.price)
)
→ write(Y.ID, Y.price)
```

This example is slightly different from the previous one. Here, the variable “y” has to track the maximum price that is greater than P, which is recorded when a 20% increase in the stock’s price occurs. The same problems of explicit variables and ambiguous notation are encountered in this example.

✎ In FTL:

```
Condition:
[x ← stock_closing.price][t ← time]
eventually [ y ← stock_closing.price && y > 1.2 * x] [ t1 ← time && t1 > t][max ← y]
eventually [y > stock_closing.price && y > max && time > t1][max←y]
Action:
write(max.ID, max.price)
```

In FTL, things become even more complex in this case. A new variable “max” has to be defined in order to store the previous maximum value, then, when a new price is reported it has to be compared to the previous maximum to see if it is still greater than P. Clearly, as the condition becomes more intricate, the number of temporal variables increases and the rule becomes quite complex.

✒ In TriggerMan:

```
create trigger New_Max_So_Far1 from stock_closing
when stock_closing.ID = 'IBM'
having  Reg_Expr_Eval( increase(stock_closing.price, " 20% ", " 1 day ")
                          → bind(t, now())
                          → *(new_high_since(t, stock_closing.price)
                       )
do    write(stock_closing.ID, stock_closing.price)
```

For this example, the temporal condition has an extra function, which can detect an increase in the maximum price. Otherwise, the condition is the same as the previous example. This example demonstrates how temporal conditions may be expressed using nested temporal functions. If the function is not already available, then users may write it and add it to the system as described in an earlier chapter.

Example (4): (Moving Average) "If the five-day moving average of the of IBM's closing prices stays the same for a period of 30 days, send a notice to the corresponding broker."

⇒ In TREPL:

```
monitor update(stock_closing) as st_cl
Steady_5_Day_Avg:
(any, * stock_closing(x, x.ID = "IBM", Count(x) <= 5), Count(x) = 5, y = avg(x),
* (stock_closing(x, x.ID = "IBM"), Count(x) <= 30, avg(x – First(x) + Last(x)) = y)

→ write("steady 5-day average for one month");
```

This example requires the calculation of a moving average. The author had a hard time trying to write a TREPL rule for this example due to the primitive operators provided by the language. This rule consists of two recursive events. The first occurs when five values of the IBM stock price have been accumulated in the variable "x" in order to calculate the five-day average. The second event occurs when the calculated 5-day average is equal to the previous one. The total combined of recursively repeating the former two-event sequence would occur within 30 days. The count function here keeps track of the number of "x" updates. When count is equal to five, the language implicitly assumes that five days have passed. This is another problem because the language assumes that the stock price is updated daily, and there are no constructs to indicate the

granularity of the time unit being used. This restricts the user to fixed operators and time granularity.

✎ In FTL:

```
condition:
[x ← stock_closing.price][t ← time]
eventually ( [x← x+ stock_closing.price][t1← time && y ← x/5 && y1 = x/5 && t1 – t = 5] ) •  eventually_within_25 (eventually ( [x← x+ stock_closing.price][ t1 ← t2, t2← time &&  y1 = x/5 && t2 – t1 = 5] ) •  [y1 == y] )
action:
write("steady 5-day average for one month")
```

FTL obviously proves to be the most difficult especially in this example. The syntax of this supposedly simple trigger looks very awkward. The number of temporal variable is blown out of proportion taking into consideration the simplicity of the temporal conditions involved in this situation. Furthermore, the author found great difficulty expressing a time-delimited operator (an operator that operates with in a time window), and hence had to assume the eventually_within_25 operator. As with TREPL, the time granularity is fixed to days in this language, which is unacceptable for practical applications. The rule's complexity seems to exponentially grow as the temporal condition of the trigger becomes more complex.

✒ In TriggerMan:

```
create trigger  Steady_5_Day_Avg from stock_closing
when stock_closing.ID = "IBM"
having holds(average(stock_closing.price, "5 day"), " 1 month")
do raise event IBM_steady_5_day_avg
```

For this example, all what the user needs to do using our temporal language is to utilize a nested temporal function which returns true if the 5-day average holds steady for one month. Again, the granularity of time here is variable and is easily set by the user, and the function operates on a time-series according to the required time window and time granularity. In fact, reading the temporal condition in our language is very close to

its equivalent English description that can be read as: “holds (the average of the stock_closing price for five days) for one month.”

From the previous examples, the reader can clearly notice the difference between our temporal language and the other two languages. The languages were treated uniformly with no bias. Furthermore, we have chosen examples that ranged from simple to moderately complex in order to cover a sample of the practical application spectrum. Due to space limitations, we have chosen the previously listed applications. However, we attempted to express other examples using the three languages and the differences were even much more vivid. Therefore, we omitted those examples in order to evade any bias accusations. Table 7-1 summarizes the results of our linguistic comparison according to nine different criterions.

One of the criterions listed in the table refers to a poll of 15 unbiased specialists. This poll was collected on different occasions by randomly selecting specialized (in computer science) individuals after showing them the previous four examples. To our surprise, all of the fifteen individuals had a consensus on the superiority of our language compared to the other two.

Table 7-1: Summery of the linguistic comparison results

Criterion	TTL (ours)	TREPL	FTL
Temporal Variables	Not required	Required	Required
Time granularity	Variable	Fixed	Fixed
Operators	Temporal Functions	Fixed set	Fixed set
Readability	Relatively easy	Poor	Poor
Language nature	Declarative	Semi-procedural	Semi-procedural
Algorithms	Moderately complex	Complex	Complex
Complexity	Hidden from users (pushed down to programmer-level)	On the user-level	On the user-level
Completeness	Incomplete set of temporal operators that can be extended according to application requirements	Closed, but complete set of temporal operators	Closed, but complete set of temporal operators
Poll of 15-unbiased specialists	Easy to use and read	Difficult to use and read	Quite difficult to use and read

CHAPTER 8
CONCLUSION

This work presents a novel approach to develop a general-purpose temporal rule processing system. Other research products have contributed greatly to this area, nevertheless, the languages proposed suffered in varying degrees from complexity and poor readability. Moreover, the implementations of such systems were convoluted because they have attempted to include all possible cases and scenarios including many that are practically rare. We diverge from the previous common track by providing a basic system that can be extended to suit any respective application. This approach reduces the complexity and cost of development and facilitates easy customization of the system. From a linguistic perspective, our temporal language is a declarative SQL-like language that is easy to read, understand and use. The temporal language constructs proposed in this work take the lead in simplicity, and stand tall among the other available temporal rule languages as illustrated in the previous chapter.

In summary, our temporal rule system presents a different way for dealing with temporal rules, and renders the following contributions:

- *A declarative SQL-like temporal rule definition language* that is easy to understand, read, and use.
- *Versatility and completeness* (with respect to a certain application) which are achieved via extensibility rather than built-in functionality, which is the common trend among most mainstream DBMS products which are moving to an ORDBMS

framework. This approach reduces the development effort substantially, and grants more freedom to application programmers. Furthermore, it allows the extension of the system to accommodate the application's needs and requirements rather than forcing use of only the built-in system capabilities. Accordingly, experts would write libraries of possibly needed temporal functions for use by mainstream application programmers. Such libraries may be specialized in specific fields of applications. For instance, the financial and trend analysis library would contain all the common temporal functions needed to write temporal triggers for applications in this specific field.

- *Simple implementation and better maintainability* due to the underlying object-relational technology.
- *Sophisticated but relatively easy to use timer management system.*

From an implementation viewpoint, a prototype of TriggerMan that includes the basic capabilities of the prescribed temporal rule system has been developed. Our temporal trigger system provides tools that are powerful and easy to use for temporal rule as well as time-series definition and manipulation. In addition, efficient and fast temporal rule evaluation is achieved using discrimination networks in conjunction with direct application of temporal functions to locally stored histories. We envision diverse utilization of our temporal trigger system, and we strongly believe that users will be able to closely interact with our system to achieve their desired goals. However, there are potential prospects to enhance our temporal rule system, which we will address as part of future work.

LIST OF REFERENCES

[Abb87] T. Abbod, K. Brown, H. Noble, “Providing time-related constraints for conventional database systems”, *In Proceedings of the 13th International Conference on Very Large DataBases (VLDB)*, pp.167-175, Brighton, England, September 1987.

[ACT96] A Joint Report by the ACT-NET Consortium, “The active database management system manifesto: A rulebase of DBMS features”, *ACM-SIGMOD Record*, 25(3):40-49 , September 1996.

[Adel97] B Adelberg, H. Garcia-Molina, and J. Widom, “The STRIP system for effeciently maintaining derived data”, *In Proceedings of the ACM-SIGMOD 1997, International Conference on Management of Data*, pp. 147-58, Tucson, Arizona, May 1997.

[Alf96] N. Al-Fayoumi, and E. Hanson, “A solution to the parallel trigger termination problem”, *Tech. Report CISE-UF TR-96-9*, CISE Department, University of Florida, 1996.

[And75] O. Anderson, “Time Series Analysis and Forecasting”, Butterworth & Co, London, 1975.

[Ari86] G. Ariav, “A temporally oriented data model”, *ACM Transactions on Database Systems*, 11(4):499-527, December 1986.

[Bern90] P. Bernstein, M. Hsu, and B. Mann, "Implementing recoverable requests using queues", *In Proceedings of the ACM-SIGMOD International Conference on Management of Data*, pp. 112-122, Atlantic City, N.J., May 1990.

[Box76] G. Box, S. Hillmer, and G. Tiao, "Analysis and modeling of seasonal time series", Seasonal Analysis of Economic Time Series , pp. 309 -334, U.S. Department of Commerce, Bureau of the Census, 1976.

[Bran93] D.A. Brant and D.P. Miranker, “Index support for rule activation”, *In Proceedings of the ACM-SIGMOD International Conference on Management of Data*, pp. 42-48, Washington, D.C., May 1993.

[Bro87] P. Brockwell, and R. Davis, “Time Series: Theory and Methods”, Springer-Verlag, 1987.

[Brow85] L. Brownston, R. Ferrell, E. Kant, and N. Martin, “Programming Expert Systems in OPS5: An introduction to rule-based programming”, Addison-Wesley, Massachusetts, 1985.

[Chak89] S. Chakravarthy et. Al., “HiPAC: A research project in active, time-constrained database management”, *TR XAIT-89-02*, Xerox Advanced Information Technology, 1989.

[Chak91] S. Chakravarthy and D. Mishra, “An event specification language (Snoop) for active databases and its detection”, *Technical Report UF-CISE TR-91-23*, CISE Department, University of Florida, September 1991.

[Chak93a] S. Chakravarthy, and D. Mishra, “Snoop: An Expressive Event Specification Language For Active Databases”. *Technical Report UF-CISE-TR-93-007*, CISE Department, University of Florida, 1993.

[Chak93b] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S. K. Kim, “Anatomy of a composite event detector”, *Technical Report UF-CISE-TR-93-39*, CISE Department, University of Florida, 1993.

[Chan93] R. Chandra and A. Segev, “Managing temporal financial data in an extensible database*”, In Proceedings of the 19th Conference on Very Large Databases (VLDB)*, pp. 302-313, Dublin, Ireland, September 1993.

[Chan94] R. Chandra, A. Segev, and M. Stonebraker, “Implementing calendars and temporal rules in next-generation databases”, *In Proceedings of the 10th International Conference on Data Engineering*, pp. 264-273, Houston, Texas, February 1994.

[Chaw96] S. Chawathe, A. Rajaraman , H. Garcia-Molina , and J. Widom, “Change Detection in Hierarchically Structured Information,” *Proceedings of The ACM-SIGMOD ‘96 Conference*, pp. 493-504, Montreal, Canada, June 1996.

[Cho95] J. Chomicki and D. Toman, “Implementing temporal integrity constraints using an active DBMS”, *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 7(4):566-581, August 1995.

[Clif87] J. Clifford and A. Croker, “The historical relational data model (HRDM) and algebra on lifespans”, *In Proceedings of the Third International Conference on Data Engineering*, pp. 528-537, Los Angeles, CA, February 1987.

[Codd70] E. F. Codd, "A relational model of data for large shared data banks", Communications of The ACM, 13(6):377-387, June 1970.

[Coup94] T. Coupaye, C. Collet, and T. Svensen, "Naos-efficient modular reactive capabilities in an object-oriented database system", *In Proceedings of the 20th Conference on Very Large Databases (VLDB)*, Santiago de Chile, Chile, pp.132-143, September 1994.

[Date93] C. J. Date and Hugh Darwen, "A Guide to the SQL Standard", 3rd Edition, Addison-Wesley, 1993.

[Day88] U. Dayal, "Active database management systems", *In Proceedings of The 3rd International Conference on Data and Knowledge Bases*, pp. 150-169, Jerusalem, Israel, June 1988.

[Day96] U. Dayal, A. Buchmann, and S. Chakravarthy, "The HiPac project", *In Active Database Systems-Triggers and Rules For Advanced Database Processing*, Morgan Kaufman Publishers Inc., Ch.7, pp. 177-206, 1996.

[DB97] "DataBlade Developers Kit, User's guide", Informix® Press, July 1997.

[Delc89] L.M.L. Delcambre and J.N. Etheredge, "The Relational Production Language: A production language for relational databases", *In L. Kerschberg, editor, Expert Databse Systems – In Proceedings of the 2nd Int'l Conference*, pp. 333-351, Benjamin/Cummings, Redwood City, CA, 1989.

[Dewa92] H.M. Dewan, D. Ohsie, S.J. Stolfo, O. Wolfson, and S. Da Silva, "Incremental database rule processing in PARADISER", *Journal of Intelligent Information Systems*, 1(2):177-209, 1992.

[Dia91] O. Diaz, N. Paton, and P. Gray, "Rule management in object-oriented databases: A uniform approach", *In Proceedings of The 17th International Conference on Very Large Databases (VLDB)*, pp. 317-326, Barcelona, Spain, September 1991.

[Dyr92] C. E. Dyreson and R. T. Sondgrass, "Time stamp semantics and representation", *TempIS Tech. Report 33*, Computer Science Department, University of Arizona, May 1992.

[Elm90] R. Elmasri, G. Wuu, and Y.J. Kim, "The time index: An access structure for temporal data", *In Proceedings of the International Conference on Very Large Databases (VLDB)*, Brisbane, Queensland, Australia, pp. 1-12, September 1990.

[Eps79] R. Epstein, “Techniques for processing aggregates in Relational database systems”, Ph.D. Dissertation, University of California at Berkeley (UCB), 1979.

[Etz92a] O. Etzion, A. Gal, and A. Segev, “A temporal active database model”, Technical Report LBL-32587, Lawrence Berkeley Laboratory, 1992.

[Etz92b] O. Etzion, A. Gal, and A. Segev, “Temporal support in active databases”, *In Proceedings of the 2nd Workshop on Information Technology and Systems*, pp. 23-38, 1992.

[Etz93a] O. Etzion, “PARDES: A data-driven oriented active database model”, *ACM-SIGMOD Record*, 22(1):7-14, March 1993.

[Etz93b] O. Etzion, A. Gal, A. Segev, “Data driven and temporal rules in PARDES”, *In Proceedings of the 1st International Workshop on Rules in Database Systems (RIDS*), pp. 92-108, Edinburgh, Scotland, August 1993.

[Forg82] L. C. Forgy, “Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem,” *Artificial Intelligence*, vol. 19, pp. 17-37, 1982.

[Gad88] S. K. Gadia, “A homogeneous relational model and query languages for temporal databases”, *ACM Transactions on Database Systems*, 13(4):418-448, December 1988.

[Gal93] A. Gal and O. Etzion, “New perspective in temporal databses”, *ISE-TR-93-6*, Technion-Israel Institute of Technology, 1993.

[Gal96] A. Gal, O. Etzion, A. Segev, “TALE: A Temporal Active Language and Execution Model”, *In Proceedings of The 8th International Conference on Advances in Information System*, CAiSE'96, pp. 20-24, Heraklion, Crete, Greece, May 1996.

[Gat91] S. Gatziu, A. Geppert, and K. R. Dittrich, “Integrating active concepts into an object-oriented database system”, *In proceedings of the 3rd International Workshop on Database Programming Languages*, pp. 395-415 , Nafplion, Greece, December 1991.

[Geh91] N. Gehani, and H. V. Jagadish, “Ode as an Active database: Constraints and triggers”, *In Proceedings of the 17th International Conference on Very Large Databases (VLDB)*, pp. 327-336, Barcelona, Catalonia, Spain, September 1991.

[Geh92a] N. Gehani, and H. Jagadish, “Active database facilities in Ode”, *Data Engineering Bulletin*, 15(1-4):19-22, 1992.

[Geh92b] N. Gehani, H. Jagadish, O. Shmueli, "Composite Event Specification in Active Databases: Model & Implementation", *In proceedings of the 17th International Conference on Very Large Databases (VLDB)*, pp. 327-338, Vancouver, British Columbia, Canada, August, 1992.

[Geh96] N. Gehani, and H. Jagadish, "Active database facilities in Ode", In Active Database Systems: trigger and Rules For Advanced Database Processing, Chapter 8, pp. 207-232, Morgan and Kaufmann publishers, Inc., 1996.

[Gor92] O. Gorawalla and A. U. Tansel, "A temporal relational database management system", Technical Report, Bilkent University, July 1992.

[Gran95] F. Grandi, M.R. Scalas and P. Tiberio, "A history-oriented temporal SQL extension", *In Proceedings of the second International Workshop on Next Generation Information Technologies and Systems (NGITS '95)*, pp. 42-48, Naharia, Israel, 1995.

[Hans92] E. N. Hanson, "Rule condition testing and action execution in Ariel", *In Proceedings of the ACM-SIGMOD International Conference on Management of Data*, pp. 49-58, San Diego, CA, June 1992.

[Hans95] E. N. Hanson and J. Widom, "Rule processing in active database systems", *Advances In Databases and Artificial Intelligence*, (1):33-77, 1995.

[Hans96a] E. N. Hanson, "The design and implementation of the Ariel active database rule system," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 8(1):157-172, February 1996.

[Hans96b] E. N. Hanson, S. Bodagala, M. Hasan, G. Kulkarni, J. Rangarajan, "Optimized rule condition testing in Ariel using Gator networks", *TR 95-027*, CISE Department, University of Florida, http://www.cise.ufl.edu, October 1995.

[Hans97] E N. Hanson and S. Khosla, "An introduction to the TriggerMan asynchronous trigger processor", Tech. Report *TR-97-007*, CISE Department, University of Florida, http://www.cise.ufl.edu, 1997.

[Hud89] S. E. Hudson, and R. King, "A self adaptive, concurrent implementation of an object-oriented database management system", *ACM TODS Transactions on Database Systems*, 14(3):291-321, September 1989.

[ILL97] "Illustra time-series support", An Illustra Technical White Paper, Illestra's Web page, http://www.informix.com, 1997.

[Info97] "Informix Dynamic Server with Universal Data Options", http://www.informix.com, 1997.

[Ish91] T. Ishida, "Parallel rule firing in production systems", *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 3(1):11-17, 1991.

[Jae95] U. Jaeger, and J. C. Freytag, "Annotated bibliography on active databases", SIGMOD Record 24(1): 58-69, 1995.

[Jok93] J. Jokiniemi and Palomäki, "Real-time Databases: A Needs Survey", *Research Report No. J-15*, Laboratory for Information Processing, VTT. Helsinki, February 1993.

[Kol89] C.P. Kolovson and M. Stonebraker, "Indexing Techniques for Historical Databases", *In Proceedings of the 5th International Conference on Data Engineering*, pp. 127-137, Los Angeles, CA, February 1989.

[Kud93] T. Kudrass, H. Branding, A. Buchmann, and J. Zimmerman, "Rules in an open database: The REACH rule system", *In Proceedings of the 1st International Workshop on Rules in Database Systems*, pp. 111-126, Edinburgh, Scotland, August 1993.

[Lee96] H. Lee, "Support for temporal events in SENTINEL: Design, implementation, and preprocessing", Masters dissertation, University of Florida, 1996.

[Lor87] N. A. Lorentzos and R. G. Johnson, "TRA: A model for a temporal relational algebra", *In Proceedings of The Conference on Temporal Aspects in Information Systems*, pp. 99-112, France, May 1987.

[Mil90] T. Mills, "Time Series Techniques for Economists", Cambridge University Press, New York, 1990.

[Mir87] D. Miranker, "TREAT: a better match algorithm for AI production systems". *In Proceedings of AAAI National Conference on Artificial Intelligence*, pp. 42-47, August 1987.

[Mot97] I. Motakis and C. Zaniolo, "Temporal aggregation in active database rules", *In Proceedings of the ACM-SIGMOD '97 International Conference on Management of Data*, pp. 440-451, Tucson, AZ, June 1997.

[Nav87] S. B. Navathe and R. Ahmed, "TSQL: A language interface for history databases", In Proceedings of The Conference on Temporal Aspects in Information Systems, pp. 113-128, AFCET, North-Holland, May 1987.

[Nav89] S. B. Navathe and R. Ahmed, "A temporal relational model and a query language", *Information Sciences*, 49(2):147-175, 1989.

[ODB94] ODBC 2.0 Programmer's Reference and SDK Guide, Microsoft Press, 1994.

[Orac97] Oracle Cartridges Reference Manual, Oracle® 1997.

[Oust94] J. Ousterhout, *Tcl and the Tk Toolkit*, Addison Wesley, 1994.

[Ris89] T. Risch, "Monitoring database objects", *In Proceedings of The 15th National Conference on Very Large Databases (VLDB)*, pp. 445-453, Amsterdam, The Netherlands, August 1989.

[Ros91] E. Rose and A. Segev, "TOODM: a temporal, object-oriented data model with temporal constraints", *In Proceedings of the International Conference on the Entity-Relationship Approach*, pp. 205-229, San Mateo, CA, 1991.

[Sar87] N. L. Sarda, "Modeling the time and history data in database systems", *In Proceedings CIPS Congress 87 Winnipeg*, pp. 15-20. CIPS, May 1987.

[Sar90a] N. L. Sarda, "Algebra and query language for a historical data model", *The Computer Journal*, 33(1):11-18, February 1990.

[Sar90b] N. L. Sarda, "Extensions to SQL for historical databases", *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 2(2):220-230, July 1990

[Seg87] A. Segev and A. Shoshani, "Logical modeling of temporal data", *In Proceedings of the ACM-SIGMOD '87 International Conference on Management of Data*, pp. 454-466, San Francisco, CA, May 1987.

[Sesh94] P. Seshdri, M. Livny, and R. Ramakrishnan, "Sequence query processing", *In the Proceedings of the ACM-SIGMOD International Conference on Management of Data*, pp. 430-441, Minneapolis, Minnesota, May 1994.

[Sesh95a] P. Seshdri, M. Livny, and R. Ramakrishnan, "SEQ: A model for sequence databases", *Tech. Report*, University of Wisconsin-Madison, 1995. Extended version of Sesh95b.

[Sesh95b] P. Seshdri, M. Livny, and R. Ramakrishnan, "SEQ: A model for sequence databases", *In Proceedings of the 11th Conference on Data Engineering*, pp. 232-239, Minneapolis, Minnesota, March 1995.

[Shu88] R. Shumway, “Applied Statistical Time Series Analysis”, Prentice Hall, Englewood Cliffs, 1988.

[Sist95a] A. Prasad Sistla and Ouri Wolfson, “Temporal conditions and integrity constraints in active databases”, *In Proceedings of the ACM-SIGMOD ’95 International Conference on Management of Data*, pp. 269-280, San Jose, CA, May 1995.

[Sist95b] P. A. Sistla and Ouri Wolfson, “Temporal Triggers in Active Databases,” *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 7(3):471-486, June 1995.

[Sko96] M. Sköld, and T. Risch, “Using partial differencing for efficient monitoring of deferred complex rule conditions”, In Proceedings of The 12th International Conference on Data Engineering (ICDE ’96), pp. 392-401, New Orleans, Louisiana, 1996.

[Snod84] R. Snodgrass, “The temporal query language Tquel”, *In Proceedings of the ACM Symposium on Principles of Database Systems*, pp. 204-212, Waterloo, Ontario, Canada, April 1984.

[Snod86] R. Snodgrass and I. Ahn, “Temporal databases”, *IEEE Computer 19*, pp. 35-42, September 1986.

[Snod90] R. Snodgrass, “Temporal databases: Status and research directions”, *ACM-SIGMOD Record*, 19(4):83-89, Reading, December 1990.

[Snod93] R. Snodgrass, “An Overview of Tquel”, Temporal Databases: Theory, Design, and Implementation. Benjamin/Cummings 1993.

[Snod92a] R. Snodgrass, I. Ahn, G. Ariav, D. Batory, J. Clifford, C. Dyreson, R. Elmasri, F. Grandi, C. Jensen, W. Käfer, N. Kline, K. Kulkarni, T. Cliff Leung, N. Lorentzos, J. Roddick, A. Segev, M. Soo, S. Sripada, “TSQL2 Language Specification”, SIGMOD Record 23(1): 65-86, 1994.

[Snod92b] R. Snodgrass, “Temporal Databases”, *In Proceedings of the International Conference on GIS: From sapce to Territory*, Volume 629, A.U. Frank, I. Campri, and U. Formentini, editors, September 1992.

[Snod94a] R. Snodgrass, S. Gomez, and E. McKenzie, “Aggregates in temporal query language TQuel”, *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 1994.

[Snod94b] R. Snodgrass, I. Ahn, G. Ariav, D. Batory, J. Clifford, C.E. Dyreson, R. Elmasri, F. Grandi, C.S. Jensen, W, Kaefer, N. Kline, K. Kulkarni, T.Y. Cliff Leung, N. Lorentzos, R. Ramakrishnan, J.F. Roddick, A. Segev, M.D. Soo, S.M. Sripada, "A TSQL2 Tutorial", *ACM-SIGMOD Record* 23(3):27-33, 1994.

[Snod95] R. Snodgrass, I. Ahn, G. Ariav, D. Batory, J. Clifford, C.E. Dyreson, R. Elmasri, F. Grandi, C.S. Jensen, W, Kaefer, N. Kline, K. Kulkarni, T.Y. Cliff Leung, N. Lorentzos, R. Ramakrishnan, J.F. Roddick, A. Segev, M.D. Soo, S.M. Sripada, "The TSQL2 Temporal Query Language", Kluwer Academic Publisher, Norwell, Massachussetts, 1995.

[Soo91] M. Soo, "Bibliography on temporal databases", *ACM-SIGMOD Record*, 20(1):14-23, Reading, 1991.

[Soo92] M. Soo, R. Snoddgrass, C. Dyreson, C.S. Jensen, and N. Kline, "Architectural extensions to support multiple calendars", *TempIS Technical Report 32*, Computer Science Department, University of Arizona, 1992.

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

[Stol91] S. Stolfo, H. Dewan, and O. Wolfson, "The PARULEL Parallel Rule Language", *In Proceedings of the 1991 International Conference on Parallel Processing (ICPP)*, pp. II/36 - II/45, St. Charles, IL, August 1991.

[Ston87] M. Stonebraker, E. Hanson, and P. K. Chan, "The design of POSTGRESS rule system", *In Proceedings of the 3rd IEEE International Conference on Data Engineering (ICDE)*, pp. 365-374, Los Angeles, CA, 1987.

[Ston90] M. Stonebraker, L. Rowe, and M. Hirohama, "The implementation of POSTGRESS", *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 2(7):125-142, March 1990.

[Ston97] M. Stonebraker, "Object-relational DBMS's: The next great wave", Informix®, with Dorothy Moore, 1997.

[Su83] S. Y. Su, "SAM*: a semantic association model for corporate and scientific-statistical databases", Journal of Information Sciences, pp.151-199, 1983.

[Tan86a] A. U. Tansel and M. E. Arkun, “HQUEL: a query language for historical relational databases”, *In Proceedings of The 3rd International Workshop on Statistical and Scientific Databases*, Luxembourg, pp. 538-546, July 1986.

[Tan86b] A. U. Tansel, “Adding time dimension to relational model and extending relational algebra “, *Information Systems*, 11(4):343-355, 1986.

[Tan93] A. U. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass, “Temporal Databases”, *Benjamin/Cummings Publishing Company, Inc.*, 1993.

[Tan97] A. U. Tansel, “Temporal Relational Data Model”, *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 9(3): 464-479, 1997.

[TSDB97] “Time-series DataBlade: User’s guide”, Informix® Press, July 1997.

[Wid91] J. Widom, R. Cochrane, and B. Lindsay, “Implementing set-oriented production rules as an extension to Starburst. *In Proceedings of the 17th International Conference on Very Large Databases (VLDB)*, pp. 275-285, Barcelona, Catalonia, Spain, September 1991.

[Wid92] J. Widom, “The starburst rule system: Language design, Implementation, and applications”, *In IEEE Data Engineering*, (15)1-4, 1992.

[Wid96a] J. Widom and S. Ceri, “Active Database Systems: trigger and Rules For Advanced Database Processing”, Morgan and Kaufmann publishers, Inc., 1996.

[Wid96b] J. Widom, “The Starburst active database rule system”, ”, *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 8(4): 583-595, 1996.

[Wied91] G. Wiedhold, S. Jajodia, and L. Litwin, “Dealing with granularity of time in temporal databases”, *In Proceedings of the 3rd Conference on Advanced Information Systems Engineering (CaiSE’91)*, pp.124-140, Trondheim, Norway, May 1991.

[Wolf91] O. Wolfson, H. Dewan, S. Stolfo, Y. Yemini, "Incremental evaluation of rules and Its relationship to parallelism", *In Proceedings of the ACM-SIGMOD ‘91, International Conference on Management of Data*, Pp. 78-87, Denver, CO, May 1991.

[Wolf93] O. Wolfson and A. Ozeri, "Parallel and distributed processing of rules by data-reduction", *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 5(3):523-530, June 1993.

[Wols96] A. Wolski, J. Karvonen, and A. Poulakka, “The RAPID case study: requirement for and the design of a fast-response database system”, *Proceedings of the 1st Worksop on Real-Time Databases (RTDB’96)*, pp. 32-39, Newport Beach, CA, March 1996.

[Wuu92] G.T.J. Wuu and U. Dayal, “A Uniform Model for Temporal Object-Oriented Databases”, *In Proceedings of the 8th International Conference On Data Engineering*, pp. 584-593, Tempe, AZ, February 1992.

BIOGRAPHICAL SKETCH

Nabeel AL-Fayoumi is a Jordanian born in Saudi Arabia in May 1968. He earned his B.Sc. in electrical engineering/computer engineering form Jordan University of Science and Technology (J.U.S.T) in 1991. He worked as a Teaching and Research Assistant in the college of applied engineering at Yarmouk University from 1991 to 1993. Through a Fulbright scholarship, he received the degree of M.E. in computer engineering from the University of Florida in 1995. He is will receive his Doctor of Philosophy degree in computer engineering from the University of Florida in August 1998. Subsequently, he will return to Jordan where he is to fill the position of an assistant professor of computer science and engineering at Yarmouk University.