Managing Temporal Financial Data in an Extensible Database *

Rakesh Chandra and Arie Segev

Walter A. Haas School of Business
University of California at Berkeley
and
Information and Computing Sciences Division
Lawrence Berkeley Laboratory
Berkeley, CA 94720
email: crakesh@csr.lbl.gov, segev@csr.lbl.gov

Abstract

Complex financial products and trading applications are difficult to model and implement in conventional commercial databases due to temporal, object, rule and other data support requirements. Extensible database systems provide a better solution because of the ability to define complex data types, manipulate data of type procedure, define rules, operators and access methods to optimize these operators. The paper discusses the design issues in modeling financial trading systems in extensible DBMS. The complexity of financial products is analyzed and strategies for modeling these products are proposed. Operators relevant to financial trading are discussed alongwith access methods to optimize these operators. The paper describes an implementation of a financial trading system in POSTGRES.

Keywords: Temporal Databases, Complex Objects, Rule Processing, Extensible Databases.

*This work was supported by an NSF Grant IRI-9116770 and by the Applied Mathematical Sciences Research Program of the Office of Energy Research, U.S. Department of Energy under Contract DE-AC03-76SF00098. *Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.*
Proceedings of the 19th VLDB Conference
Dublin, Ireland, 1993.

1 Introduction

Since 1971, after the breakdown of the Bretton Woods[MANK92] system of fixed exchange rates, financial markets have seen a sharp increase in the fluctuation of interest rates and exchange rates. The past 20 years have also seen rapid advances in information technology that make it possible to collect and process large amounts of data. These events, when coupled with the sophistication of financial theory, have created a marketplace of a vast array of financial products that cater to different investment needs. Financial Trading Applications, which are meant to facilitate trading in these products, have also become very complex because :

- There are a variety of complex financial products available and the dynamic nature of the market, coupled with a decrease in regulation, has created a situation in which new products are introduced often and old ones discontinued.
- Trading strategies are based on numerically intensive procedures and complex mathematical relationships between financial products.
- The decrease in the cost of telecommunications and the increased reliability of networks make profitable trading opportunities available only for short periods of time.

The proliferation of financial products, increase in information and rapid advances in technology keep trading houses and investment firms under constant pressure to develop new ad-hoc applications for financial trading support. These applications tend to be expensive, involve the duplication of effort to a large extent and are for the most part, product specific. The investment is often wasted if the product is discontinued. In addition, the non-uniformity of applications makes it very difficult

for the firm to get a clear picture of its overall risk at any point in time.

Commercial databases are unable to handle the complexity of financial products and trading applications because they are specialized for the creation, manipulation and processing of fixed-format snapshot records rather than temporal data. Extensible database systems [CARE87] provide an appropriate environment for the development of high-performance financial trading applications. The primary goal of this paper is to describe the complexity of financial data and focus attention on strategies for designing trading applications using extensible databases. While current extensible database system prototypes support many features required for the implementation of such applications, this paper identifies requirements that are essential but lacking in current systems.

1.1 Relevant Research

Rapid advances in technology have changed the way investment firms do business. This has been well documented in [BAUE92] and [SPEC88]. [BAUE92] discuss the impact that computers have had on the ability of investment firms and trading houses to quickly analyze numerous trading strategies. [SPEC88] discuss the role computers and trading systems had in the Oct. 19, 1987 crash. The presence of databases in trading systems was discussed in [ABBO88], [PEIN88] and [SAMM87]. [ABBO88] point to the response time requirements of a database that facilitates stock-trading. [PEIN88] and [SAMM87] present real-life experiences gained from a study of a large, high volume stock trading system that used a standard relational DBMS.

The main focus of this paper is to demonstrate how extensible databases can be used to build reliable, high-performance financial trading applications. There are many prototype extensible systems including POSTGRES [STON90a], Exodus[CARE88], Starburst[LIND87] and Ode[AGRA90]. A more complete discussion of the capabilities required by trading systems and the supporting features available in these prototypes is presented in Section 3 and Section 4.

[SEGE87] provided a convenient way to look at temporal data through the concepts of *Time Sequence* and *Time Sequence Collection*. Temporal data models are designed to capture the complexities of many time-dependent phenomena. Temporal data modeling and representation have been extensively studied in the literature in [CLIF87], [GADI88], [SEGE88a], [SNOD87], [WUU92]. A glossary of temporal concepts can be found in [JENS92]. This paper models temporal objects found in the financial domain and discusses the implementation of the model in an extensible database.

The rest of the paper is organized as follows. Section 2 presents two examples of financial data and related trading applications. This section highlights the complexity of the data and the difficulty of designing trading applications using traditional databases. It also describes the functionality necessary to support a trading application. Based on the examples presented in Section 2, Section 3 gives an outline of the design of a financial trading application using an extensible DBMS. This section also discusses the functionality available in existing object-oriented and extensible databases with reference to the functionality required for this application. Section 4 shows how a trading application can be designed and implemented in POSTGRES Section 5 summarizes the contributions of the paper and discusses issues for further research.

2 Trading Applications and Financial Data

This section introduces finance concepts and terminology necessary to understand the functionality of a financial trading application. It highlights the complexity of financial data and the related application. The functionality required of a database that can support financial trading is also discussed. It should be noted that for this paper, the terms *financial product* and *financial instrument* are synonymous.

Term Structure

A debt instrument is a promissary note that evidences a debtor/creditor relationship. In such a relationship, one party borrows funds from another party and the borrowing party promises to pay the funds, together with interest. An example of a debt instrument is a US Govt Treasury Bond (T-Bond). The length of time till the debt instrument matures is called its *term-to-maturity* or *term*. Since there is a chance that the borrower will fail to make timely payments of interest and/or principal on the debt, the associated risk is quantified and called the default risk.

Each bond of a given term is priced by the market and this price can be converted by valuation arithmetic into a yield. The relationship between yield and term is called the term-structure of interest rates. When graphed, it is known as a yield curve. The yield curve/term-structure varies over time due to the fluctuation of interest rates. Traders, financial product designers and economists are interested in the shape of the yield curve at a particular point in time (cross-sectional data) and also in the changes in the yield curve over a period of time (time-series data).

If the term-structure were to be modeled in a database, the following database features are required :

- Capability of creating data types of arbitrary complexity using base data types and other complex data types. An example of a complex data type is the data type time-series which is a 2 dimensional array of float and time. The term-structure at time t, is a data structure of type time-series. The term-structure over a period of time is a complex data type and is a 2 dimensional array of time-series and time. This feature allows the creation of hierarchical structures that specify relationships between different data objects. This includes the capability to specify different kinds of term-structures as instances of the object *term-structure*, e.g., US-Govt-Treasury term-structure, Corporate term-structure.
- Capability of defining procedures that can take objects of type time-series as arguments, e.g., draw-yield-curve(term-structure).
- Capability of defining rules that specify relationships between objects.

In addition to the database features mentioned above, other features needed to model complex financial objects are :

- Capability of defining rules that act as constraints on the values of attributes of an object.
- A rule syntax that allows the expression of events based on the state of objects in the database, external events, time-based events, triggering of other rules and the execution of procedures.
- Capability of expressing temporal conditions in rules.
- Capability of supporting data of type procedure.
- Active rules that check for conditions rather than only being event driven.
- Capability of specifying the sequence of rule execution.

An example of a financial contract is now discussed to illustrate the requirements mentioned above.

Option Contract

A *call option* on an underlying asset, say a stock, grants its purchaser the option to purchase a specified number of shares of the stock from the seller of the option. If the option is classified as a European Option, this right is good till the expiration date. For simplicity, this is the only case discussed. The *life* of an option is the time between the date of purchase and the expiration date. The price at which the purchaser can buy the stock is called the *exercise* price and will be denoted as K. To buy this right, the purchaser must pay an option *premium*. The stock price at any point in time will be denoted as S, and if $S > K$, then the option is said to be in-the-money. If $S = K$, the option is at-the-money and if $S < K$, the option is out-of-the-money.

STOCK OPTION CONTRACT	
Attribute	Rules
Asset	Stock Symbol, e.g., BMW
Size	50 shares of the underlying stock
Expiration Months	Next Three Months as well as closest 2 quarterly expiration months. The quarterly expiration months are March, June, September, December.
Exercise Price	For each expiration month, there must be at least 3 Call Options at least one exercise price in-the-money, at least one exercise price at-the-money and at least one out-of-the money. Exercise prices separated by DM 5 intervals.
Exercise Period	Last Trading Day of the option
Last Trading Day	The third Friday of the expiration month if that day is an exchange trading day, otherwise the exchange day preceding this Friday
Expiration Day	The exchange trading day following the last trading day
Settlement Day	Two exchange days after the exercise

Figure 1: Part of an Option contract

Figure 1 shows the attributes of an option contract that is traded on the Frankfurt Stock Exchange and the rules governing the instantiation of these attributes. These attributes include the *underlying stock* and the *size* which are easily modeled in a database as text and integer respectively. The *last trading day* is an attribute that represents the last day on which the option can be bought or sold on the exchange. This is derived from the expiration month based on the following rule "the third Friday of the expiration month if that is an exchange trading day, otherwise the exchange trading day immediately preceding this Friday". In this rule, *exchange trading days* refer to the days on which trading occurs on the Frankfurt Stock Exchange. The rule clearly demonstrates the need for a concise language to express temporal conditions and the capability of the database to understand these rules and manipulate temporal objects. The database must also give the user the capability of defining collections of time points, e.g., syntax to define the set of time points that

constitute the exchange trading days.

An implicit part of the contract is its cash flow pattern. At the boundary points, the cash flow calculation is simple and is given by the following rule :

```
On date the contract is bought
       cash flow = - (premium + transaction costs)
On expiration date
       if (Stock price of underlying asset (S) >
   exercise price (K)) then
        cash flow = S-K
       else
        cash flow = 0
```

Cash flow patterns can be very complex and are usually expressed in a high level programming language. A risk-profile is a two-dimensional array that records the change in cash flow with changes in the price of the stock. It is generated from the cash flow pattern and the database should be able to either maintain the risk-profile which entails supporting objects of the type 2 dimensional float array or maintain the logic to derive the risk-profile. Maintaining the logic to compute the risk-profile and cash flow pattern can be accomplished by supporting data of type procedure and rules that trigger these procedures based on certain conditions.

The correct procedure to value an option contract is determined by the cash flow pattern during the life of the option. This procedure is called the *valuation scheme*. Valuation schemes are essentially modules of code written in high-level programming languages. Since the valuation scheme is dependent on the cash flow pattern, the database must be able to store the valuation scheme associated with the option and execute this procedure when necessary. An additional requirement is that the database rule schema must ensure that all parameters necessary for the valuation scheme, e.g., stock price volatility and risk-free interest rate, are estimated before the valuation scheme is triggered.

In addition to the rules described above and in Figure 1, the database must also store rules that define relationships between financial contracts. If there is a mathematical relationship between the value of two or more financial contracts, it can be defined as a rule in the database. A violation of this rule opens up a potentially profitable arbitrage opportunity. (Arbitrage is defined as the simultaneous buying and selling of equivalent portfolios to obtain a riskless profit.) Since such opportunities aren't available for very long, the rule should have a constraint associated with it. This constraint will either recheck the arbitrage condition before committing the transaction of buying and selling the two portfolios or have a maximum bound on the time that the transaction can take.

In this section, we discussed the complexity of financial objects and the related trading application. The next section discusses the design of a financial trading application using an extensible database.

3 Trading Applications using Extensible Databases

The discussion in the previous section makes it clear that the design of a trading application requires a database that provides data management, object management and knowledge management capability. Object management entails efficiently storing and manipulating complex data types. Knowledge management refers to the capability of storing and enforcing rules to reflect the semantics of the application [STON90b]. Extensible DBMSs have this capability because :

- They provide the capability of adding complex data types to the base types of *float*, *int* and *char*. This allows complex financial products to be modeled in the database, thus creating a uniform and central store for financial products and financial data.
- They provide the facility to declare new operators on base and complex data types. This allows the definition of procedures that can take financial products as arguments. An example is an operator to compute the term-structure from the database of bonds. Section 4 discusses two such important operators.
- They provide facilities for implementing new access methods designed to optimize user-defined operators. This feature allows the creation of indexes that can facilitate operators on financial products. One such index is discussed in Section 4.
- Rules can be defined and their execution sequence controlled. This allows knowledge of the financial application to be built into the system. If rules are processed efficiently applications can avoid the expense and performance problems due to inefficient application programs. Relationships between financial products can be expressed and active rules created to facilitate arbitrage trading. Rules also make it possible to construct alerters that can be used to call the attention of traders to unusual activity or an important news item.

Extensible DBMSs also provide a natural environment for building trading applications because a building blocks approach to application development can be adopted. This approach preserves investment in software by encouraging the modular design of applications, promoting generality in the design of modules, allowing reuse of existing modules and enforcing a standard interface for modules. The following discussion focuses on important elements of trading applications and database modeling issues.

Elements of a Trading Application

The important elements of a trading application that are to be modeled in a database are : (a) temporal objects, e.g., time-series such as the price of a stock over time, (b) cross-sectional objects, e.g., objects whose time-varying characteristics are not recorded by the database, (c) rules, e.g., expression of the fact that the value of an option must be recomputed when the stock price changes by some pre-specified amount, (d) methods or procedures, which are essentially modules of code developed in high-level programming languages such as C++. Examples of methods are valuation schemes for options and procedures for computing the term-structure. Both rules and methods add domain knowledge to the database and reduce the amount of application code required. The other elements are (e) calendars that describe sets of time points like a particular date/time or time intervals like years and (f) external objects which are convenient abstractions for objects that are not defined in the application. An external object could be an externally updated information source such as a stock ticker or data feed coming from other locations not running the same application.

It should be noted that the above classification is not a disjoint partition of the elements of a trading application. For example, a rule could change over time. This makes the rule a temporal object as well. A class is a collection of these objects. For example, the option contract described in Section 2 could be modeled as a class consisting of temporal objects, rules, methods and cross-sectional objects.

The following discussion provides a detailed specification of features that must be built into an extensible DBMS. The most important of these features are calendars, temporal objects and rules.

Calendars

The discussion of option contracts in Section 2 highlighted the need for a powerful language and algebra to express natural language time-based expressions. Financial trading applications must also be able to trade on a global basis. Trading around the world requires knowledge of the different trading days and trading hours on different exchanges.

A system of calendars and relational operators is used to achieve this functionality. [SOO92] first introduced extensible calendric systems. Calendric systems are collections of calendars and operators. They also discuss a toolkit that allows the definition of new calendars and calendric systems. Our modeling of calendars uses the simple set based algebra defined in [LEBA86]. Operators for these calendars include interval based relational operators like *overlaps*, *meets*, *precedes* and operators that facilitate deriving calendars from other calendars. Calendars and the related operators are useful for (a) defining the time points at which temporal objects have values, (b) defining the temporal logic for rules that have triggers based on time, e.g., rules that alert traders that certain options are to expire at time T or in the interval $[T_s, T_e]$, (c) describing sets of time points or time intervals, e.g., AMERICAN-BUSINESS-DAYS $\equiv$ DAYS-IN-YEAR − WEEKENDS − HOLIDAYS, (d) defining constraints, e.g., suppose no new option contracts can be introduced if any current options are to expire in 10 exchange trading days. To express this rule, calendars are used with appropriate operators to express the set of days, *exchange trading days*. Then relational operators will operate on the derived calendar to express the logic of the rule. Examples are presented in Section 4, (e) representing natural language time-based expressions, e.g., the 3rd Friday of the month, (f) allowing different semantics for date arithmetic, e.g. the yields on some bonds are computed based on the actual number of days between two dates but with the assumption that the year always has 360 days and (g) maintaining valid time in databases.

This system of calendars allow a concise representation of time points and intervals and make it unnecessary to physically store time points associated with temporal objects. Thus, it is imperative for data manipulation operators, that have temporal objects as arguments, to work in close association with the calendar system.

In Section 4 we discuss the implementation details of calendars, relational operators and a data manipulation operator. The discussion also presents the time algebra used to denote time-based natural language expressions.

Temporal Objects

Temporal objects track the environment over time. It should be emphasized that a temporal object is a generic term for both a simple time-series and complex temporal objects. A time-series can be considered to be a "sequence of values in the time domain for a single entity instance" [SEGE87], e.g., stock prices. A collection of n-ary vectors grouped together to represent a semantic unit is also a temporal object but will be referred to as a complex temporal object. An example of a complex temporal object is a Company Balance Sheet. The individual items like Gross Sales, Cost of Goods Sold and Operating Expenses are time-series that are recorded at the same time points and are reported together. Rules and procedures that have been changed over time are also considered to be temporal objects. These general semantics of temporal objects allow us to consider versions as a special case of temporal objects.

Since it is possible to construct complex temporal objects from sets of simple time-series, the discussion will

focus on time-series. Time-Series modeling and representation are an integral part of modeling complex financial objects and will be discussed in detail below. Part of the following functionality can be supported by [ROSE91] and [WUU92].

Each time-series is essentially an n-ary vector and is associated with a set of user-defined information. This information (M) is classified into (a) information that must be present with every time-series (M_s) (if not supplied by the user, appropriate defaults are used) and (b) information optionally supplied by the user (M_u). M_s consists of :

1. Name : The identifier of the time series to be used in data retrieval and data manipulation routines.

2. Calendar/Granularity : a set of pre-defined time points. This item specifies the calendar with which the time-series is associated. For example, the time-series IBM-DAILY-CLOSING would be associated with the calendar AMERICAN-BUSINESS-DAYS. This means that on every day in the calendar AMERICAN-BUSINESS-DAYS, the time-series should have a value. Granularity is a specification of the points in time in the defined calendar that can potentially have data values [SEGE87]. The defined calendar will thus determine the granularity of the time-series. The advantage of associating a time-series with a calendar is that there is no need to physically store the individual time points with the values of the time-series. When the time-series is retrieved due to a query, the individual time points can be generated using the specification of the calendar. This is especially advantageous for time-series with large lifespans. Since the individual time points are not saved on disk, there are large savings in disk space utilization. Thus, all time points of the time-series are physically stored only when the calendar cannot be pre-defined. This is possible in the case of randomly updated time-series like tick-by-tick stock prices.

3. Exception-Set : is a set of time points (within the calendar) on which values of the time-series are not recorded. For example, even though IBM-DAILY-CLOSING should be recorded on every day in the calendar AMERICAN-BUSINESS-DAYS, there may be an important announcement on a particular day that stops trading in the stock. Thus the value of the time-series is not recorded on that day. The exception-set will include such time points. Thus, the actual calendar for a time-series is the set difference of *Calendar* and *Exception-Set*.

4. Lifespan : This indicates the start time and end time of the time-series. The end time can be specified to be ∞. The lifespan is used in conjunction with the calendar and exception-set to generate the set of time points for which the time-series has values.

5. Update Mode : This indicates whether the time-series is derived from another time-series(s) or is base data. If the series is derived, the rule for update is specified here. Time-series are allowed to have a hybrid update mode. For example, a time-series recording the value of an option will change whenever the price of the underlying stock changes (price is derived) and also when the option is traded on the market. In the latter case, the price is not derived and is determined by the value at which the option was exchanged on the trading floor. A more detailed treatment of the Update Mode is provided in [ETZI92].

6. Frequency : This specifies the frequency with which the time-series is updated. The time of update refers to the valid time. Valid time is defined in [JENS92] as "the time when the fact is true in modeled reality". Frequency is always specified with respect to the calendar with which the time-series is associated and may be a non-trivial function on the set of time points in this calendar. For example, suppose EMP, a time-series which records the level of employment in the country, has the Calendar/Granularity : "the last day of the month unless the day is a holiday in which case it is the preceding business day". The frequency of EMP would be *monthly*. If a time-series is derived from other time-series, the frequency would be the frequency of the base data or some function of it. For example, consider the time-series DJIA and DJIAHILO. DJIA, the Dow Jones Industrial Average, is a weighted average of the price of a given set of stocks. It is computed every time the price of a component stock changes. Thus, it is a derived time-series with the same frequency of update as the base data. On the other hand, DJIAHILO, which is a time-series that contains the daily high and low values of the DJIA, has a daily frequency which is different from the frequency of its base data.

   It is important to stress the difference between frequency and granularity. A time-series is said to be "regular" [SEGE87], if it contains a value for each time point in the time-series lifespan. In a regular time-series, the granularity is the same as the frequency. In this case, the exception-set is a null set.

7. Type [SEGE87] : The type of a time-series determines how to derive values of the time-series at time points where the value isn't explicitly specified.

An example of a one-dimensional time-series vector with the associated user-defined information is the ob-

servations of a country's Gross National Product (GNP). The calendar associated with GNP is a function of the AMERICAN-BUSINESS-DAYS calendar. There is no exception-set defined for this time-series. GNP is not derived from any other time-series and thus its update mode is "Base Data". The frequency of update is quarterly and reflects the dollar value of the sum total of economic activity in the quarter. The type of the time-series GNP is user-defined. This means that user-defined functions will be used to determine the value of GNP at time points where it has not been explicitly recorded. For example, the GNP on April 30th (valid time) is not recorded in the time-series. This could be derived by a function which uses the previous values of GNP or through a function which uses other economic indicators. Rules (discussed below) can be used to define the type of a time-series and build in the desired level of complexity. A detailed treatment of time-series modeling in databases can be found in [SEGE92].

Rules

Rules are useful for testing integrity constraints, maintaining consistency, versioning, materialized views, updating derived data [STON90b] and monitoring the database for specific events [DAYA88]. In the framework of the trading application, the functionality demands that a rule be a 6-tuple $< Rule - List, Calendar, Event - Condition, Action, Transaction - Coupling, Constraint >$. Each component of the 6-tuple is explained below :

- Rule-List : This is a collection of rule ids and is used to group rules that must be executed in sequence. The position of a rule within the rule list determines the order of execution.
- Calendar : The calendar associated with a rule defines the time points/intervals when the rule is active. As noted in the discussion on calendars, this gives the user unlimited flexibility in specifying time-based rules. For example a rule can be fired at a specific point in time, e.g., on Wednesday at 10 a.m., at certain intervals of time, e.g., every 5 hours, at specific points in time, e.g., Mon, Wed, Friday, and always, in which case they become active rules. The default value for the calendar is *null* which means that unless specified, rules will be event-driven.
- An Event's scope is defined as the set of objects that determine the occurrence of the event. Thus, the scope can be a set of rules (for rule-triggered events), methods (for events based on the execution of procedures) and calendars (for time-based events). The condition can be based on the current state of database objects or historical states. The event-condition that triggers a rule is specified by using elements of the events' scope, a condition and logical connectives like *and*, *or*, *exclusive-or*. For example, a trigger can be based on the state of an object and time.
- Actions are either rules or methods. Thus, actions can trigger other rules, execute methods, update database objects and perform any database functions that can be done through a method. Actions are also allowed to update rules. The utility of this functionality in trading applications is illustrated by the following example. In times of great uncertainty, it would make sense to compute the term-structure often. Thus the rule for computing the term structure would have the syntax "Every T minutes do compute-term-structure", where T would have a small value. But in times of lower volatility, an active database would update the value of T, so that the term-structure is computed less often. This frees up system resources for use in other tasks.
- Transaction Coupling : This defines the coupling between the event and action in the rule. A transaction is an ordered set of methods and rules bounded by a begin transaction and commit/abort. Based on this definition of a transaction, four types of coupling can be defined [GEHA92] (a) immediate : action is executed immediately after the event is recognized in the same transaction. This is the default value. (b) deferred : The action is executed just prior to the commit of the transaction that recognizes the event, (c) dependent : The action is executed as a separate transaction but only after the transaction that recognizes the event has committed, and (d) independent : The action is executed as a separate transaction with no dependency on the transaction that recognizes the event. The latter transaction could abort or commit without affecting the action.
- Constraint : This is a simple Rule (see below) or calendar and is used to enforce timing constraints on the execution of rules. If the constraint is a calendar and if the transaction-coupling is either immediate, deferred or dependent, it indicates the maximum time that can elapse between the beginning of the event transaction and the commit of the action transaction. If the time constraint is not met, the action is aborted. A simple rule will reexecute the event transaction and check the result of the event with the value that was previously obtained. The default value of the constraint is *null*, which means that no constraint is applicable to the rule unless specified.

In this section, we have provided a detailed specification of the important elements of a trading application. The following section discusses the choice of POSTGRES as the extensible database to implement our ideas and describes important aspects of the implementation.

4 Implementation

Although recent work in temporal databases [ROSE91], [ROSE93], [SU91],[WUU92], describe very useful functionality not available in existing prototypes, they are not fully implemented and integrated with other features such as abstract data types and extensible access methods. Consequently, we limited the implementation alternatives to those discussed below. We employed five criteria in choosing an extensible system. These were (a) support for rules, (b) preference for a model that was an extension of the relational paradigm, (c) presence of a *fast path* capability that allowed the creation of indexes to optimize any operators that we defined, (d) persistent programming language access and (e) availability. Our four alternatives were Ode, Starburst, Exodus and POSTGRES. We soon realized that the general rule capability required for the trading application was not available in any data model and that only POSTGRES would allowed us to modify the source code to implement this capability. Ode provides persistent programming language access though O++ but the non-availability of the source code and the fact that it is based on the C++ object paradigm made it an unattractive alternative. Starburst[HAAS90] is an extensible relational DBMS that provides the capability to create complex objects, new storage methods, optimization of new operators, specification of the storage method for tables and a general rule capability [WIDO92]. We found the fast path capability more difficult to use compared to POSTGRES. Exodus[CARE88] includes two basic components - The storage object manager which provides concurrent and recoverable access to object of arbitrary size and the type manager that has a set of base types which can be extended by users. Exodus also provides libraries of database system components for access methods and version management. It provided the *E* implementation language and a generator that produces a query optimizer and compiler from the description of the available operations and methods. Between POSTGRES and Exodus, we chose the former because of the availability and the fact that we had worked with this model before and understood the design and implementation well. Also Exodus provided no basic rule capability.

Overview

POSTGRES is a next generation extensible relational DBMS with general mechanisms that can be used for semantic data modeling. These mechanisms include (a) abstract data types which are used to support complex objects (b) data which can be of type procedure and (c) rules.

The primary goal was to implement calendars, temporal objects and rules. The implementation of temporal objects is done by using the POSTGRES feature of declaring complex data types. Calendars are implemented by using stored procedures and user-defined operators. The POSTGRES Rule System is not adequate for the demands of the trading application. It must be extended to include (a) event specification that includes time-based events, triggering of rules and/or execution of procedures, (b) decoupling the action part of the rule from the event if specified, (c) ordering the sequence of execution of a set of rules and (d) imposing time constraints and rechecking the event condition before committing the action transaction. POSTGRES allows user-defined operators and access methods. Operators relevant to this application were defined and appropriate access methods designed to optimize these operators. For brevity only calendars and access methods are discussed in this section.

4.1 Calendars

The implementation of calendars involved creating the data type interval and set of intervals, which are called Calendars. The algebra on which the implementation is based was formally introduced in [LEBA86].

A *Calendar* is formally defined as a structured set of intervals and the *Order* of a calendar is defined as a measure of the depth of the structured set. Thus, the set $S = \{(l_1, u_1), (l_2, u_2), \cdots, (l_n, u_n)\}$ is a calendar with order 1 while $R = \{S_1, \cdots, S_m\}$ where $S_i = \{(l_j, u_j)\}$ is a calendar of order 2. A calendar of order 0 is simply a set of numbers.

A set of basic calendars, e.g., YEARS, HOURS, were created as system defined calendars and the relationships between them were expressed in a table which had the format, $\{Calendar_1,\ Calendar_2,\ \textit{list}\}$. Here $Calendar_i$ is a text variable and the *list* is an order 0 calendar. For example, to express the relationship between YEARS and MONTHS, the entry in this table would be {YEARS, MONTHS, 12}, which means that 12 MONTHS ≡ YEARS. The relationship between YEARS and DAYS is more complicated because of a leap year every 4 years. Consequently, the entry in the table is : { YEARS,DAYS,(365,365,366,365)}, which means that in the first year there are 365 days, the second year has 365 days and so on as the list specifies for four years, after which the same pattern is repeated. The relationships between the system defined calendars are based on a start date which was taken as Jan 1, 1970 (the start date on

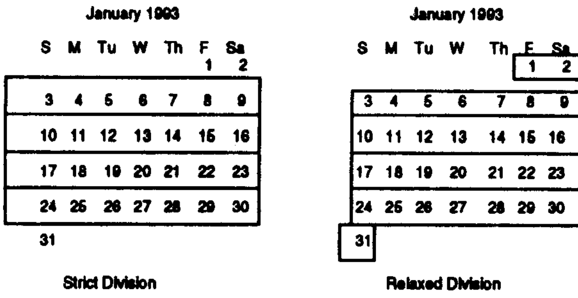

Figure 2: Result of strict and relaxed division

the UNIX system).

To create a new calendar two different operators were defined. The first operator is called *generate* and takes the arguments ***start time***, *end time* and a list of numbers. The operator *generate* creates the following calendar :

$$generate(T_s, T_e; int_1; \cdots; int_n) =$$
$$\{(T_s, T_s + int_1), (T_s + int_1, T_s + int_1 + int_2), \cdots,$$
$$(T_s + \Sigma_{i \leq n} int_i, T_s + \Sigma_{i \leq n} int_i + interval_1), \cdots\}$$

where T_s is the start time and T_e is the end time. In *generate* the list of numbers is considered a circular list and the operation is carried out till the end time is exceeded. The operator, *generate*, is illustrated with the following example.

$$YRS-SINCE-1987 = generate(Jan-1-87; Jan-3-92;$$
$$365, 366, 365, 365) = \{(1, 365), (366, 731),$$
$$(732, 1096), (1097, 1461), (1462, 1826), (1827, 1829)\}$$

where the second element in the calendar, $\{(366, 731)\}$, denotes that the second year, 1988, began 366 days from Jan 1, 1987 and ended 731 days from Jan 1, 1987. It should be noted that January, 1, 1987 is taken as 1.

The second operator to create a new calendar is based on the same logic as the system table that stores relationships between system defined calendars. This operator is called *caloperate* and it takes the arguments calendar and a list of intervals. The operator, $caloperate(C, T_e; x_1; x_2; \cdots; x_n)$, where C is the calendar from which the new calendar is to be derived, would create a new calendar whose first interval is a union of the first x_1 intervals of calendar C, the second interval is the union of the second x_2 intervals of C and so on. The list is applied as a circular list. *caloperate* is illustrated by the following example. If YEARS is the system defined calendar $\equiv (1, 365)$, then $caloperate(YEARS, *; 7)$, would give the calendar of weeks in the year since :

$$(YEARS, *; 7) \equiv \{(1, 7), (8, 14), (15, 21), \cdots, \}$$

Here $*$ indicates an arbitrary end time.

Generically, a relational operator (Op) takes two intervals to generate a third interval. POSTGRES is easily extended to support relational operators like ***intersection*** *cover*, *overlaps*, *during*, *meets*, *contains*, $<$ and $\leq$ operators since the semantics of these operators are well defined. Two new operators, the ***division*** operator and *selection* operator, were introduced to facilitate manipulation of calendars. The division operator takes a calendar of order 1 as its left argument, an interval as its right argument and generates a calendar of order 1 as the result. If the right argument is a calendar, then it operates on every interval in the calendar.

For each relational operator, there are two interpretations of the division operator. Formally the strict division (:) operator is defined as :

$$C : Op :< t_s, t_e > \equiv \{c \cap < t_s, t_e > |(c \in C) \wedge (c \ Op < t_s, t_e >)\}$$

The relaxed division (.) operator is defined as :

$$C.Op. < t_s, t_e > \equiv \{c | (c \in C) \wedge (c \ Op \ < t_s, t_e >\}$$

where the interval $(-\infty, \infty)$ is excluded from the resulting sets. If weeks in the year 1993 are :

$$\{WEEKS \equiv \{(-4, 2), (3, 9), \cdots\}$$

and $\{Jan - 1993 = \{(1, 31)\}\}$, the elements in the calendars $WEEKS$: *during* : Jan-1993 and $WEEKS.during.Jan$-1993 are illustrated in Figure 2.

The operator *selection*, denoted by $[x]/C$, selects the xth interval from the calendar C. The operator *recursive-selection*, denoted by $[x]_r/C$, selects the xth interval from the calendar C recursively, till the result is an order 0 list. Specifically if C is an order 2 calendar, the xth interval is chosen from each element. If C is an order 1 calendar, the xth element of each interval is chosen.

4.2 Operators and Access Methods

POSTGRES provides the capability of defining operators, written in high-level programming languages, to the database and then using them in the query language. In this section, we describe three operators relevant to financial trading. The first operator, *ComputeTS* operates on the database of bonds to create the term-structure. The *Transformation* operator converts a time-series from its current frequency to another frequency. POSTGRES also allows a fast path capability to access its internals for creating new indexes and access paths to optimize the user-defined operators. The access method used to optimize the *ComputeTS* is described in detail. An algorithm for the *Transformation* operator is also discussed.

In *ComputeTS*(**B**,Range,Default,Category), **B** indicates the set of bonds that are to be used to compute

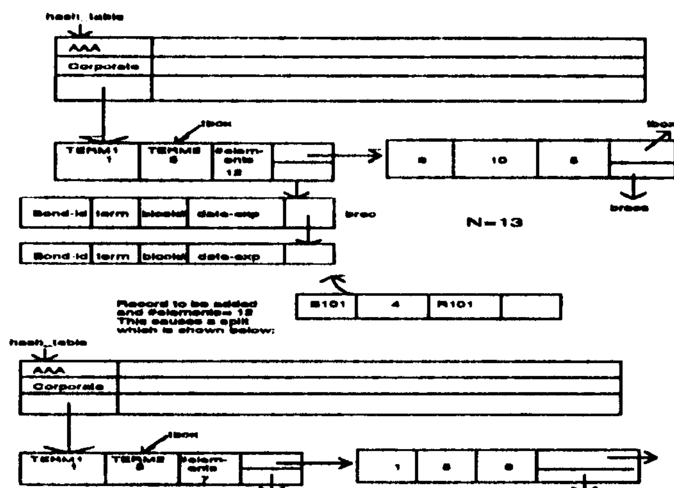

Figure 3: Adding records to Main-Memory Index

the term-structure. *Range* is the range of time between which the term-structure is to be computed. *Default* indicates the default risk for which the term-structure is computed and is determined by the investment grade rating assigned to it by credit rating agencies. *Category* indicates the type of bonds to be used in the computation, e.g., Corporate, Government, Mortgage.

The computation involves the steps of selecting the set of bonds that have a given default risk and category type from among the space of bonds, **B**. If **B** is unspecified, the entire database of bonds is used and for each bond in the previous set, computing the yield, i, by using the price of the bond, the vector of cash flows, **CF**, and the expiration date of the bond. To compute the yield of a bond, the remaining term, the cash flows and the current bond price are required. This operator returns a two-dimensional array of yield and term.

From the formulas described above, it is clear that the computation of the term-structure is a numerically intensive procedure. It is further complicated by the fact that the fluctuation of bond prices on a minute-to-minute basis requires recomputation of the term-structure very often. Thus, the data structure that is used to optimize the computation of a term-structure must allow (i) parallel computation of parts of the term-structure, (ii) the choice of the degree of accuracy required in term-structure construction such that as the accuracy desired decreases, the response time improves and (iii) computation of only a small portion of the term-structure with an improvement in response time. The main memory data structure shown in Figure 3 is proposed to optimize the computation of the term-structure.

The *hash-table* contains all possible combinations of bond-rating and bond category. A hash-function, $h()$, is appropriately chosen to avoid any collisions between these combinations. Each element in the hash-table points to a list of buckets. Each bucket is characterized by a remaining term range (RTM). A bucket with RTM 1-5 would contain bonds of a specific category and bond rating with remaining term between 1 and 5 years. Each bucket record contains information on a particular bond including its remaining term, date of expiration and cash flow if the cash flow is simple. It also contains a pointer to the disk block where the bond details are physically stored. Information on the number of elements in the bucket is also maintained dynamically so that the buckets can be reorganized when necessary. The advantage of this data structure is that it allows term-structures of different ranges to be computed by different processors. For example, if the term-structure for bonds of AAA rating and the category *corporate*, is to be determined for the range 1-30 years and two processors are available, the job could be divided among the two by sending a snapshot of the data structure for 1-15 years to one processor and the snapshot for 16-30 years to the other. Partial term-structures will be computed by each processor and the result is concatenated. In addition, this data structure allows the construction of a term to a user-specified degree of accuracy.

The data structure must be updated whenever (a) a new bond is issued and (b) a bond expires/defaults or is called (forcibly expired by the issuer) and (c) periodically to correctly reflect the RTM. Buckets are allowed to grow a maximum of N, after which they are split and the records distributed equally over the new buckets. Figure 3 shows the case when a new bond record is added and the number of records in the appropriate bucket is 12. Since $N = 13$, it causes the creation of a new entry in the bucket list and an equal redistribution of records between the two buckets. Buckets with the same RTM are recombined only when the number of records in each bucket is less than $\lceil N/m \rceil$. When the number of records in contiguous buckets with different RTM have less than $\lceil N/m \rceil$ each, and neither bucket has an adjacent bucket with the same RTM, the buckets are coalesced and the RTM of the resulting bucket is expanded. The values of N and m are dependent on the frequency of insertions and deletions in the index.

The operator, *Transformation*(TS,F), is used to convert time-series from the existing frequency to another. TS is a time-series and F is the frequency to which TS is to be converted. Conversions from a lower frequency to a higher frequency are allowed only if the semantics of the transformation are clear. Since the time points of a time-series are not explicitly recorded, the transformation operator uses the calendar, exception-set, lifespan and frequency of a time-series to convert it into a time-series with another frequency. To express the transformation operator algebraically the following definitions are introduced :

- The set of system defined calendars is CO.
- $domain(CO)$
- $rel(C_1, C_2)$ indicates the relation between calendars C_1, C_2, e.g., $rel(MONTHS, DAYS) = (DAYS, *; 31, 28, \cdots)$
- $rank(c)$ is used to assign an ordering to the system defined calendars,
  e.g., $rank(SECONDS) = 1$ and $rank(HOURS) = 3$.
- $hcc(c_1, c_2)$, the highest common calendar, is formally defined as $hcc(c_1, c_2) = \{c_i | \ rank(c_i) \geq rank(c_j),$
  $\forall c_i, c_j, \ (rel(c_1, c_i) \wedge (rel(c_2, c_i) \wedge (rel(c_1, c_j) \wedge$
  $(rel(c_2, c_j)) \wedge (i \neq j)\}$.
  For example, $hcc(DECADE, YEARS) = YEARS$.
- $cal(frequency)$: maps every frequency to a system defined calendar.
  Formally, $\{cal(frequency) \mapsto c | c \in CO\}$.
- Given these definitions, we are now in a position to formally define the *transformation* operator.

  $$Transformation(TS, F1) = \{TS(i), \forall i \in ([n]_r / COLL)\}$$

  where,

  $$\{COLL = ((TS.C) \ .overlaps.$$
  $$rel(cal(F1), hcc(cal(TS.F), cal(F1))))\}$$

  where $TS.C$ and $TS.F$ are the calendar and frequency of the Time-series that is being transformed. $TS(i)$ is short form for the value of the time-series at the time point i and $[n]_r$ is short form for the selection operator that selects the last item from the calendar. Note that this selection operator is applied recursively till the calendar is an order 0 calendar.

This is illustrated with the following example : A time-series with weekly frequency and the calendar AMERICAN-BUSINESS-WEEKS is to be transformed into a time-series with a monthly frequency. Since there is no simple relation between the calendars WEEKS and MONTHS, the algorithm for the *Transformation* operator is used. Using the definition of transformation and the system-defined functions, we have $cal(monthly) = MONTHS$, $cal(weekly) = WEEKS$, and
$hcc(MONTHS, WEEKS) = DAYS$.

$$rel(MONTHS, DAYS) = (DAYS, *; 31, 28, 31, 30, \cdots)$$

and $(TS.C)$ =AMERICAN-BUSINESS-WEEKS, which is a calendar of weeks in the year.

$$\{(1,5), (8,12), (15,18), (22,26), (29,33), (37,40),$$
$$(43,47), (50,54), (57,61), \cdots, \}$$

Then $\{(TS.C) \ .overlaps.(DAYS, *; 31, 28, 31, 30, \cdots)\}$ results in a calendar of order 2 :

$$\{\{(1,5), (8,12), (15,18), (22,26), (29,31)\}$$
$$\{(32,33), (37,40), (43,47), (50,54), (57,59)\}, \cdots, \}$$

From this calendar, the last interval is selected and results in an order 1 calendar : $\{(29,31), (57,59), \cdots, \}$. Since the selection operator will be applied recursively till the calendar is of order 0, the last element is selected from each interval, resulting in an order 0 calendar : $\{31, 59, \cdots, \}$. The transformed time-series would then be the value of TS at each time point in this calendar.

5 Conclusions and Further Research

Fluctuations in interest and exchange rates, rapid advances in information technology and financial theory, have created a marketplace of a vast array of complex financial products. Financial trading applications, which are meant to facilitate trading in these products have also become very complex. Because of the constant pressure to keep up with the market, investment firms and trading houses are forced to create product specific trading systems that are discarded as soon as the product is discontinued. To preserve the investment, a building blocks approach to application development should be adopted. Extensible database systems provide an environment for developing fast high-performance applications. The main objective of this research is to focus attention on strategies for designing trading applications using extensible databases.

The contributions of this paper include (a) analysis of the complexity of financial products and design of strategies for modeling them in extensible databases. This includes a complete specification of temporal objects, rules and calendars, (b) introduction of operators relevant to financial trading, (c) design of access methods to optimize these operators and (d) implementation of the design in POSTGRES.

We are looking at the following areas for further research:

- Introduction and optimization of financial trading operators. The paper described two important operators relevant for financial trading and access methods to optimize these operators. A detailed study of financial trading should suggest the basic operations that can be used as building blocks for more complicated operations. We are compiling this list of operators, so that appropriate access methods for the optimization of these operators can be developed.

- Storage methods for temporal objects encountered in trading applications is an open problem. There have been several proposals in the literature for efficient storage and retrieval of temporal and multidimensional data but it is not clear which proposal is

the best or whether a completely new approach is required. We are currently doing a performance analysis of storage structures based on typical queries that are encoutered in financial analysis.

References

1. [ABBO88] Abbott, R. and Garcia-Molina, Hector, "Scheduling Real-time Transactions," Sigmod Record, Vol. 17, No. 1, March 1988, pp.71-81.
2. [AGRA90] Agrawal, R. and Gehani, N. H., "ODE (Object Database and Environment): The Language and Data Model," *Proc. ACM SIGMOD 1989*, Portland Oregon, 1989, pp.36-45.
3. [BAUE92] Bauer, R. J. and Liepins, G.E.,"Genetic Algorithms and Computerized Trading Strategies," in "Expert Systems in Finance", O'Leary, D.E., and Watkins, P.R. ed., Elsevier Publishers, 1992, pp.89-100.
4. [CARE87] Carey, M (ed.), "Special Issue on Extensible Database Systems," Database Engineering, June 1987.
5. [CARE88] Carey, M., et.al. "The Architecture of the EXODUS Extensible DBMS," in Readings in Database Systems, Stonebraker M. 1990, Morgàn Kaufman.
6. [CLIF87] Clifford, J. and Croker, A. "The historical relational data model HRDM and an algebra based on lifespans", in *Proc. Third International Conference on Data Engineering*, pp. 528-537, Los Angeles, February 1987.
7. [DAYA88] Dayal, U., et. al., "The HiPAC Project: Combining Active Databases and Timing Constraints," *Proc. ACM SIGMOD Record* Vol. 17, No. 1, March 1988.
8. [ETZI92] Etzion, O., Gal, A., Segev, A., "Temporal Support in Active Databases," *Proc. of the* 2nd *Int. Conf. on Information Technology and Systems,*
9. [GADI88] Gadia, S.K., "The Role of Temporal Elements in Temporal Databases," Data Engineering Bulletin 7, pp. 197-203, 1988.
10. [GEHA92] Gehani, N., Jagadish, H. V., Shmueli, O., "Event Specification in an Active Object-Oriented Database", *Proc. of ACM SIGMOD 1992*, pp. 81-90.
11. [HAAS90] Haas, L., et. al., "Starburst Mid-Flight: As the Dust Clears," IEEE Transactions on Knowledge and Data Engineering, Vol. 2, No. 1, Mar. 1990.
12. [JENS92] Jenson, C.S., Clifford, J., Gadia, S.K., Segev, A., Snodgrass, R.T. , "A Glossary of Temporal Database Concepts," ACM SIGMOD Record , Vol 21, No. 3.
13. [LEBA86] Leban, B., McDonald, D., and Forster, D., "A Representation for Collections of Temporal Intervals," in *Proc. of the AAAI-1986*, 5th Int. Conf. on Artificial Intelligence, pp. 367-371, 1986.
14. [LIND87] Lindsay, B., "A Data Management Extension Architecture," *Proc. ACM SIGMOD 1987*, San Francisco, CA, 1987.
15. [MANK92] Mankiw, G., "Macroeconomics," Worth Publishers, New York 1992.
16. [PEIN88] Peinl, P., and Sammer, H., "High Contention in a Stock Trading Database: A Case Study,", in *Proc. ACM SIGMOD 1988*, May 1988, pp. 260-268.
17. [ROSE91] Rose, E., and Segev, A., "TOODM - A Temporal, Object-Oriented Data Model with Temporal Constraints," *Proc. of the* 10th *Int. Conf. on the Entiry-Relationship Approach*, San Mateo, CA, 1991.
18. [ROSE93] Rose, E., and Segev, A. "A Temporal Object-Oriented Algebra and Data Model," Forthcoming in ECOOP93.
19. [SAMM87] Sammer, H., "Online Stock Trading Systems: Study of an application," in Proceedings of Spring COMPCON 87, San Francisco, pp. 161-163.
20. [SEGE87] Segev, A., and Shoshani, A. , "A Logical Modeling of Temporal Databases," in *Proc. ACM SIGMOD Conference* 1987.
21. [SEGE88a] Segev, A., and Shoshani, A., "The Representation of a Temporal Data Model in the Relational Environment," *Lecture Notes in Computer Science*, Vol 339, M. Rafanelli, J.C. Klensin, and P. Svensson (eds.), Springer-Verlag, pp. 39-61, 1988.
22. [SEGE92] Segev, A. and Chandra, R., "A Data Model for Time-Series Analysis," in *Proc. of Workshop on Current Issues in Databases and Applications*, Rutgers University, Sept. 1992.
23. [SNOD87] Snodgrass, R. , "The Temporal Query Language TQuel," *ACM TODS*, Vol 12, No. 2.
24. [SOO92] Soo, M., Snodgrass, R., Dyreson, C., Jensen, C.S., and Kline, N., "Architectural Extensions to Support Multiple Calendars", TempIS Technical Report 32, Computer Science Department, University of Arizona. Revised May 1992.
25. [SPEC88] Voelcker, J., et. al., "How computers helped stampede the stock market," in SPECTRUM, Oct, 1988.
26. [STON90a] Stonebraker, M., "The Implementation of POSTGRES," IEEE Transactions on Knowledge and Data Engineering, March 1990.
27. [STON90b] Stonebraker, M., Jhingran, A., Goh, J., Potamianos, S., "On Rules, Procedures, Caching and Views in Data Base Systems," *Proc 1990 ACM SIGMOD Conf. on Management of Data*, June 1990.
28. [SU91] Su, Y.H.S., Chen, H. M., "A Temporal Knowledge Representation Model $OSAM^*$/T and its Query Language OQL/T", *Proc. of* 17th *Int. Conf. on Very Large Databases*, pp. 431-442, September, 1991.
29. [WIDO92] Widom, J., "The Starburst Rule System: Language Design, Implementation, and Applications," in Data Engineering, Vol.15, No.1-4, Dec. 1992.
30. [WUU92] Wuu, G.T.J, and Dayal, U. "A Uniform Model for Temporal Object-Oriented Databases," 8th *Int. Conf. on Data Engineering*, pp. 584-593.