A Multidatabase System for Tracking and Retrieval of Financial Data

Munir Cochinwala
Vertek Software
Summit, NJ 07901

John Bradley
Independent Consultant
NY, NY 10016

Dow Jones Telerate[†]

Abstract

We have built a multidatabase system to support a financial application that stores historical data used by traders to identify trends in the market. The application has an update rate (append-only) of 500 inserts per second and also has sub-second response requirements for queries. A typical query requests between 100-1000 records. In this paper we define the characteristics of the application, the multidatabase system we used to support the applications and the extensions we made in the application to achieve the required functionality and performance.

1 Introduction

Financial applications require access to both real-time and historical data. Historical data is defined over an interval. The data in an interval can be all of the real-time data over that interval or a summary of the data over the interval. Historical data is used by traders in analysis and charting to identify trends in the market.

The real-time data concerning equities, bonds, options, mutual funds and currencies originates at various stock exchanges and brokerage houses. It is sent over multiple real-time feeds, MarketFeed [6] and Ticker [7] that can deliver data in compressed form at the rate of 56 Kilobits per second (Kbps) or 19.2 Kbps. The data is sent is on per instrument basis. An instrument is identified as an entity that has a price and is capable of being traded. All issues on exchanges are identified as instruments. The data for an instrument that is sent over a real-time feed could be the price of the instrument, bid or ask for the instrument, option price for the instrument or an actual trade for an instrument.

We have implemented a multidatabase system [13, 5, 1, 4, 3, 8, 10] that supports tracking and retrieval of historical data. The system is part of a larger project, the Platform [2], that provides access to real-time data, historical data and value-added calculations (user-defined or programmed) over the different types of data.

Storage and retrieval of historical data poses interesting database problems: most of the data is append-only, the arrival rate of the data is very high (greater than 500 ticks per second), there is a burst of data at every endpoint of any interval that is being tracked, consistency of the data is serializability on <*time, instrument*> pair, retrieval is typically for greater than 100 records and for the most recent intervals, distribution of data per database is based on load balancing, and most queries are simple selects.

The diversity of our requirements precluded use of any proposed or existing multidatabase. We did not need ACID properties for our applications. We used main memory to improve performance when durability was not needed. We did not require global serializability and in some cases, did not even need local serializability. Our consistency criteria were application-defined.

This application incorporated fundamentally different DBMS's into a multidatabase to exploit the unique attributes of each DBMS, while presenting the appearance of a single entity to users. The application

† Work was performed when both authors were at Dow Jones Telerate.

Proceedings of the 20th VLDB Conference Santiago, Chile, 1994

benefited from the strengths of each DBMS, as each DBMS's traded performance for functionality to a different extent. For instance, we used a main memory DBMS for performance and an Indexed Sequential File Manager for persistent storage in the presence of rare updates. Our multidatabase system 'cooperated' with the application to meet the performance and consistency requirements of the application.

The rest of this paper is organized as follows: Section 2 describes the model and requirements. Section 3 describes the application architecture including the functionality implemented by the application and the multidatabase system. In the next section, we describe how we met application requirements by using both the multidatabase system and the application components. The last section is the conclusion.

2 Model and Requirements

The fundamental architecture of the Platform uses the producer/consumer model. A producer is a generator and exporter of data feeds, and a consumer is an importer and user of data feeds. A process or a set of processes may be a producer or a consumer or be both a consumer and producer. The historical data application (History Engine) is both a consumer and producer. It consumes data from the real-time feeds and delivers data over an interval to consumers (traders) or other producer/consumers.

An individual Platform may have one or several sites. The history engine and the multidatabase system may be distributed across several sites of the Platform. In Figure 1, we show the architecture of the Platform. The real-time feeds can deliver compressed data at the rate of 56 Kilobits per second (Kbps) or 19.2 Kbps. Data coming from the real-time feeds are ticks, baselines, or correction. A tick is either a trade of an instrument, a bid for an instrument or a ask for an instrument. A baseline is a message received when a significant event such as an exchange open or close occurs. It is the image of an instrument that consists of the instrument name, type and current price. A correction is a message to correct an erroneous tick. The Platform is currently capable of handling 500 ticks a second.

The History Engine is also required to handle 500 ticks a second. The universe of instruments consists of 500,000 instruments. The updates for instruments generally follow a 90-10 rule. 10% of the instruments are 'hot', i.e., 90% of the updates are for 10% of the instruments. The History Engine tracks data over the following interval: ticks, 1 minute, 5 minutes, 1 hour, daily, weekly, monthly and annually. The data for ticks is simply price and volume coming off the real-time feeds. This is true for both trades and quotes. For any other interval besides ticks, the data is a summary consisting of

open, close, high, low, volume, tickcount

for that interval. The summary data is calculated from the ticks that come off the feeds. Users can request data based on a particular interval or a summary over a particular interval.

User requests are not restricted to the intervals that are tracked. A user request can be for any interval. To request historical data, a user requests by instrument name, interval, start time and end time. There is a utility program that allows ending times to be in the future. Users can request inventory of available historical data or the actual historical data. The inventory table of available data is replicated in a memory resident database for fast access. The inventory table is replicated across all the sites of the History Engine.

Users can also request creation of historical data based on patterns. The request with patterns can be for instruments that match the pattern both in the present and in the future. For instance, a user can request (for present and in the future) that historical data for an instrument with the pattern 'IBM%' be created . Subsequently, when a new option on IBM is created [1] the History Engine should 'track' it.

A stock split specifies a ratio that determine the equivalent value of the stock. For instance, a stock split of 2 for 1 means that 2 new shares are equal to 1 old share. In other words, the price of the stock is halved. The History Engine is also required to keep track of any stock splits and return data to users adjusted or unadjusted for stock splits.

2.1 Transaction Types and Consistency

In the History Engine, two types of update transactions that which have different properties and need different notions of consistency.

1. Distributed Transaction

   In the History Engine, the distributed transaction updates a replicated table: the inventory table. These transactions have to be serialized. For the inventory table, we use distributed certification as in [1]. Distributed certification ensures that the local orders are compatible with a global serial order [10].

2. Append Only

   For each instrument, appends have to be serializable in <*instrument, timestamp*> order. The

[1] All options based on IBM have 'IBM' as the prefix.

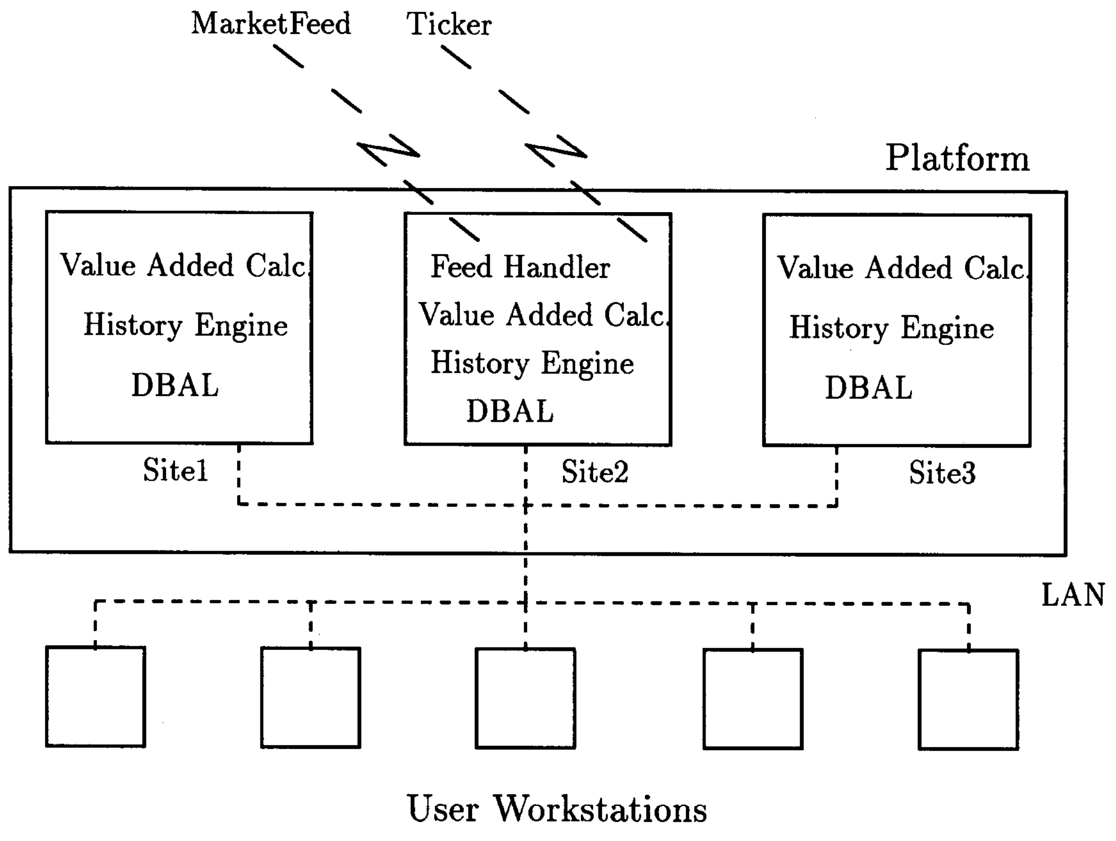

Figure 1: Platform Architecture

timestamp is generated by the source, the exchanges. These transactions are dependent only on instrument and timestamp for that instrument. Transactions for different instruments are commutative.

To maintain the serializability in <*instrument, timestamp*>, we guarantee that a single instrument will reside in a single database and insert a new batch of ticks after the previous batch has been committed to the database. An optimization for both insertion and retrieval is given in a later section.

3 Application Architecture

The functionality of storing and retrieving financial data is divided between the Database Access Layer (DBAL) and History Engine components. DBAL implements relational, multidatabase semantics using several types of commercial DBMS's and provides location transparency for all Platform data, a uniform API to all DBMS's, atomic commitment of distributed transactions and enforcement of global consistency. The History Engine performs extensions and performance optimizations based upon the unique semantics of managing historical data.

3.1 Multidatabase Functionality

DBAL enables access to several different databases via a uniform API: a relational database (InterBase)^{TM}, a main memory database (Smallbase) [14], and an ISAM file system (C-Tree)^{TM}. The ISAM file system provides a subset of database semantics. Each DBMS offers a different mix of functionality versus performance. All database operations are syntactically defined via a SQL-like language, although the semantics of each operation is limited by the functionality of the target DBMS. For example, an application would view an ISAM file as a relation via DBAL. However, complex relational operations, such as a join, would fail on ISAM files, because the operation is not supported by the file system. For our application, we do not need joins across databases and we did not implement a distributed join. The following data manipulation operations are defined: select, update, insert and delete. Data definition operations are create database, delete database, create table, delete table. Transaction man-

^{TM}InterBase is a trademark of Borland International.
^{TM}C-Tree is a trademark of FairCom.

agement functions are start transaction, commit and abort.

Data is exchanged between DBAL and the application in attribute/value pair format (ie. attribute_name = value). For example, a tuple from the history inventory relation would be represented as "InstrumentName='IBMEquity', Start=1/1/70, Interval='1 Day'".

Location transparency is provided by a single global namespace encompassing all sites and databases. Since we require that a relation name be unique across the entire Platform, an application can query a table without knowing the site or database in which it is located. A transaction can span multiple databases on different sites; two phase commit is used to guarantee atomicity.

In addition, DBAL supports user defined triggers. A trigger is a rule that consists of *event, condition and action.* Triggers in databases often are expressed by rules [11, 15, 12] defined using languages such as relational query languages and object-oriented languages [11]. Supported events are changes (insert, update, delete) to particular attributes of a relation. The condition consists of a conjunction of one or more equality expressions or a NULL expression that is always evaluated as true. DBAL supports user notification of the detected event. We do not support database operations as triggered actions.

3.2 History Engine Implementation

The History Engine isolates the application from the particulars of the database model used to store historical data by defining the concept of a history track. A track is identified by instrument name, time of the first and last stored interval, and the duration of each interval. The History Engine maps from track schema to the relational schema that it created in DBAL and vice versa. Applications access historical data exclusively by reference to track definition.

The History Engine is organized into the following discrete components to perform the task of accumulating summary data and interfacing with applications and the database. In Figure 2, we show the architecture of the History Engine.

- Tracker

  The Tracker provides the interface between the real time feeds and the history engine. Upon system startup, the Tracker will register for delivery of real time data for each of the instruments that are currently tracked.

- Time/Data Compression (TDC)

  The TDC library incorporates data from the real time data feeds (via the Tracker) into the interval summaries for the current time interval. Upon completion of an interval, the summary will be written to the database and supplied as input to larger time intervals. For example, a five minute interval will be formed from the data in the included one minute intervals.

- Data Manager (DM)

  The DM library interfaces with DBAL to read and write tracks and inventory and performs translation from the History Engine schema to database schema and vice versa. DM provides a level of abstraction from the specifics of the particular DBMS being used.

- Local History Server (LHS)

  LHS accepts requests for history tracks and inventory stored on the local Platform. LHS forwards requests to DM and returns data to the application.

- Remote History Server (RHS)

  RHS provides an interface to history servers that may be accessed via wide-area networks.

Only a single instance of the following component resides on a Platform:

- Global Tracking Manager (GTM)

  GTM selects a Tracker to accept ticks for a particular instrument based on dual criteria: load balancing across all Platform sites and maintaining consistency of all tracks for the instrument being placed.

3.3 DBAL Implementation

The implementation of DBAL uses several different types of processes (Client, Agent and Server) interacting within and across Platform sites. Client processes can access the databases by calling functions in the DBAL run-time library, which forward database requests to Agent processes to be executed. A single Server process performs resource allocation on each site. In Figure 3, we show the DBAL architecture.

- Agent

  Agents execute database operations on behalf of Clients. Each Agent is linked to the run-time library of a particular DBMS and only executes requests for that DBMS. A fixed number of Agent processes are forked during system startup and are assigned to one Client at a time on a need basis.

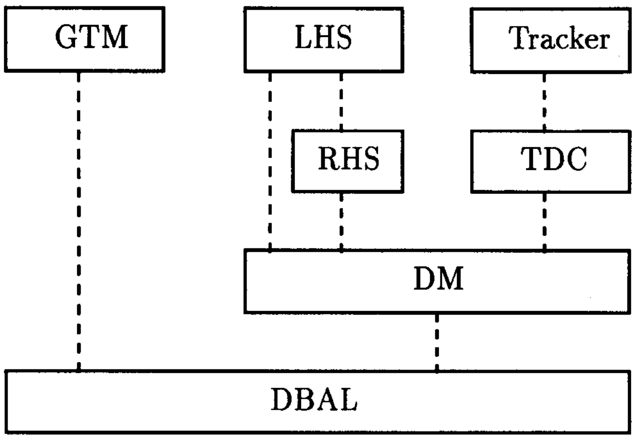

Figure 2: History Engine Architecture

When a Client requests access to a particular database, UNIX™ Inter-Process Communications will be used to link the Client and Agent process. Each Client's request is translated into the particular syntax of the target DBMS and DBMS output is translated to attribute/value pair format.

An individual Agent performs transaction management for local transactions and interacts with the coordinator as part of the two-phase commit protocol for transactions across multiple DBMS's.

- Server

  One Server process resides on each site. The Server forks the Agent processes during system startup, manages the assignment of Agents to Clients and maintains DBAL meta-data.

  The meta-data consists of relations describing each Platform database, relation, client and authorization. Each server stores a persistent copy of meta-data for local data; when a remote server is restarted, the meta-data is transferred to that site. Any changes to the meta-data is broadcast to all servers. The meta-data is used to authorize all Clients' requests and locate an agent process to execute the request. If a local agent is required, the Server will notify the selected Agent to connect to the Client. When the necessary Client is remote, a message is sent to the Server on that site to request assignment of the Agent.

- Client

  Clients may transparently access multiple, distributed databases via a uniform API by calling the DBAL run-time library. The library does not access any database directly, but it forwards database requests to Agent processes. In addition, the library interacts with the Server to authorize Client's requests and locate Agents for those request.

  The library coordinates transactions across multiple databases using two-phase commit. Coordination requires interacting with all Agents participating in the distributed transaction to guarantee that all local transaction conclude consistently.

  The DBAL library supports multiple threads, using the DCE threads package. Therefore, several client threads may access the database concurrently by individual DBAL connections. Each thread executes a separate transaction and is assigned a different agent process.

- Triggers

  Triggers are implemented by a Trigger Handling Daemon located on each site. Each handler maintains persistent data defining each trigger and each client registered to receive notification that the trigger fired. Upon startup, this data is read and kept in main storage.

  Each change to the database is forwarded to the Trigger Daemon from the Agent executing the change. Agents also forward the completion status of each transaction: commit or abort. If the change matches one of the defined rules, a notification message will be queued until the transaction commits. If the transaction aborts, the message will be discarded. Since a trigger condition cannot span multiple sites, no coordination between Daemons is required.

  Trigger definitions are structured to minimize evaluation time. Rules are organized in a hier-

™UNIX is a trademark of XOPEN.

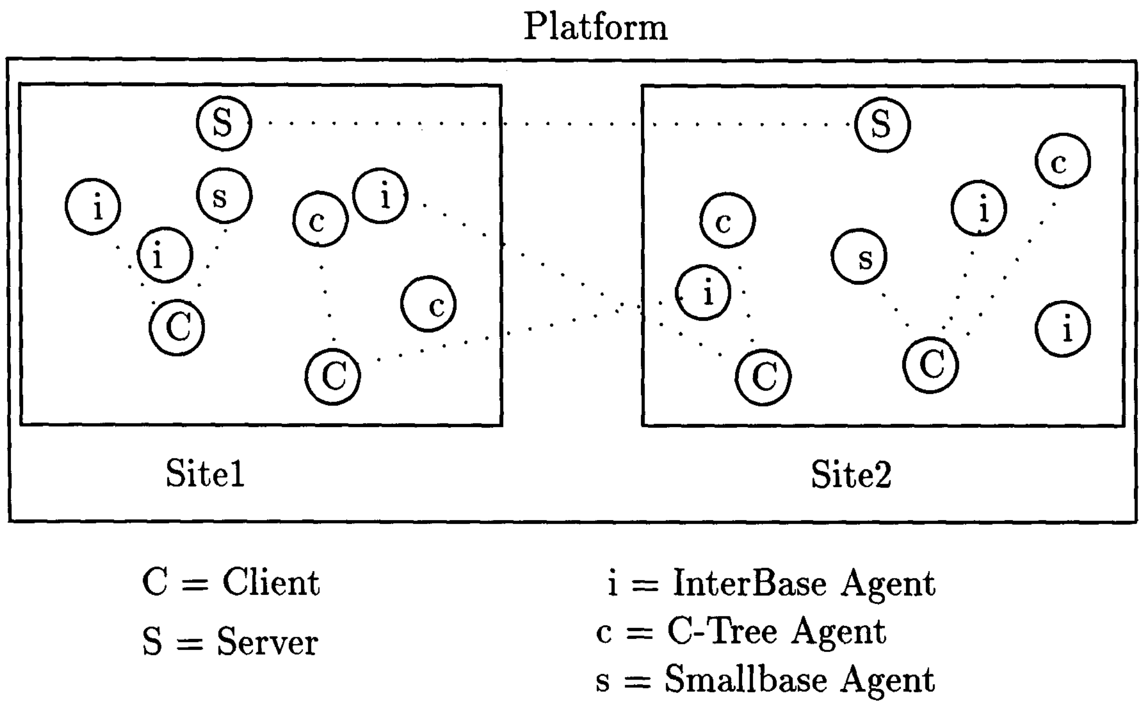

Figure 3: DBAL Architecture

archy of modification type, table name, conditional expression and trigger identifier. Within each level of the hierarchy, data is maintained in collating sequence. Therefore, changes that will not fire a trigger may be discarded at the earliest possible time. When a particular expression is used to define multiple triggers, that expression need only be evaluated once for all triggers.

4 Problem Resolution and Extensions

The multidatabase system aided us in building the application. However, we had to add extensions both to the multidatabase system and the application to adequately meet functionality and performance.

1. Consistency using Load Balancing

   Global serializability for the append-only transactions is not needed because of our algorithm for load balancing and insertion order. Our notion of correctness for an instrument is serializability on <*instrument, timestamp*> pair. We ensure this consistency is maintained by always inserting instruments in timestamp order.

   We also guarantee that a single instrument will not be spread across multiple databases. We achieve this by requiring creation of a history track to follow a protocol. A particular track is sent to a specific tracker and database based on the following rules:

   (a) If a tracker has an existing track for the instrument in the request, then assign that tracker to 'track' the new request.

   (b) If no tracker has an existing track for the instrument in the request, then choose a tracker with the least load.

2. Insert Batching

   To handle the heavy update rate, 500 updates (inserts) a second and to have the ability to retrieve 1000 points of an instrument with sub-second response, we had to put intelligence in the application to aid the DBMS in placement of the data. Recall, that each update (insert) may be for a different instrument.

   If the DBMS clustered indices and wrote in place, then the update rate may be too slow since the DBMS may have to write to 500 distinct blocks. If the DBMS buffered the ticks and wrote blocks to disk, then the retrieval rate may be too slow since the DBMS may have to access many blocks. If the DBMS wrote to a write-ahead log and applied the ticks to the appropriate disk block, then it is possible that spikes may occur in the read/write rate since in the financial environment, a slow update rate may not occur for a whole day.

As the ticks arrive, we buffer them in memory. We actually insert the ticks in the memory resident database. The ticks are written to disk based on two rules.

(a) The tick count of an instrument reaches a pre-determined value. All records for this instrument are written to the disk as a transaction to place them on a few disk blocks. This mechanism is used for instruments that are 'hot'. Recall that a 'hot' instrument is in the 10% of instruments for which 90% of the updates occur.

(b) A timer expires. All instruments that have not been written to disk since the last timer expiration are written to disk. This mechanism is used for instruments that have few ticks.

The application will execute all queries for an instrument both against the memory resident database and the disk database.

3. Triggers for Future Patterns

   Users can request creation of a track by giving patterns for instrument names. The request can be for all instruments that currently match the pattern or all instruments that match the pattern now or in the future. Patterns can consist of a wildcard suffix. A typical example is a request for instruments that match the pattern 'IBM%' now and in the future. All options that are based on IBM have 'IBM' as the prefix.

   To accomplish creation of inventory on future patterns, a trigger as implemented in DBAL is defined. When a new instrument is created matching the the pattern, a notification is sent to GTM which in turn creates the track.[2]

4. InterDay versus IntraDay

   To achieve performance gains by reducing conflicts on instrument and disk or database resources, we separate interday data from intraday data. At the end of a trading day, we move the data for that day to a separate database that is within the scope of DBAL. This database will be used mostly for reads since all inserts are done for a particular day.

   Updates can be applied to the previous day's database since corrections can occur but corrections for a previous day are quite infrequent.

[2]All instruments are listed in a separate database that is part of DBAL but beyond the scope of this paper.

5. Time versus Space Tradeoff

   Most requests are for standard intervals like ticks, 1 minute, 5 minute hourly, daily and so on. We support creation of tracks only for the standard intervals. We have defined tables for these standard intervals and can access them directly.

   However, users can request any intervals when they query the data. We have routines that can build a track for any requested for any requested interval from the standard intervals. Requests for non-standard intervals like 7.5 minutes have to calculated by going through a filter when the request is executed.

5 Conclusion

We have built a multidatabase system to support an application that has requirements for sub-second response requirements for queries that require 100-1000 points of data and had an update rate (append-only) of 500 inserts per second. The update rate peaks at the endpoints of intervals. We have also incorporated a memory-resident database for performance.

We had two different notions of consistency both defined by the application requirements. One was for the replicated table and another was for instruments. We used the knowledge of the application for the data in the instrument table (via load balancing) and distributed certification for the replicated table. Although distributed certification may be slow, we only need it when an inventory track is created or deleted. It is not needed for the append-only updates or large datasets for queries. These are restricted to a single database and in some cases, the request can be satisfied from the memory-resident database.

Currently, the system is operational on a UNIX environment supporting two disk-resident databases, InterBase and C-tree. We also have Smallbase as a memory-resident database in DBAL. The DBAL server, DBAL agents and the history processes are UNIX processes. The history processes are threaded and make multiple connections to DBAL. The processes communicate via shared memory on the same site and via sockets across sites.

The requirements for performance optimization emerged during the development cycle. Initially, we were under the assumption that load balancing may be enough to solve the performance problem since data was distributed across multiple sites. However, if we wrote a single transaction to disk every second with 500 inserts within the transaction and each insert belonging to a unique instrument, then a retrieval for 1000 points of an instrument may have to access 1000 blocks. We had to use clustering in memory to optimize the placement of updates. We have to analyze the performance of our clustering algorithm.

We found triggers to be extremely useful in creation of tracks for future inventories. Without triggers, we would have to implement a process that monitored instrument creation and did pattern matching. In the future, we need to be able to allow users to create inventory based on predicates. We also need to explore the use of triggers to do automatic roll-ups from smaller to larger intervals.

We need to evaluate the performance of the whole system. This includes the cost of a transaction in memory, the cost of distributed certification and the cost of a query going across multiple sites because of the load balancing we used. We need to also look into the notion of synthetic securities, i.e. a security composed of multiple securities and analyze how load balancing and consistency are effected.

Aside from performance, we need to introduce recovery for the entire system. Currently, we recover from a tick log that is maintained both by Platform processes and the originating Exchanges. We need to explore the correctness of the database and the historical data when the real-time feeds are down, a database crashes or a particular site is unavailable. We also need to explore dynamic load balancing of history tracks (when a tracker or site goes down) and its effect on correctness as well as performance.

Acknowledgements

We would like to thank John Sui for his support of the project and giving valuable insight for performance enhancements. We would also like to thank Ashwin Bhatt for his continual guidance and support. We would also like to thank the other implementors of the system, Lloyd Fernandes, Bruce Holenstein, Ram Tanamy and Raghu Subramanian. Finally, we would like to thank John Bopp, Rick Busdiecker and Jeff Carvalho for multiple discussions that improved the system and application.

References

[1] M. Cochinwala, K.C. Lee, W. Mansfield, Jr., and M,Yu. A Distributed Transaction Monitor. *Proc. of RIDE-IMS 1993*, April 1993.

[2] Munir Cochinwala, John Bradley, Ram Tanamy and Raghu Subramanian. A Multidatabase Solution for a Financial Application. *Proc. of Applications of Databases*, June 1994.

[3] P. P. Chrysanthis and K. Ramamritham. A Formalism for Extended Transaction Models. *Proc. of the 17th International Conference on VLDB*, September 1991.

[4] U. Dayal, M. Hsu, and R. Ladin. Organizing Long-Running Activities with Triggers and Transactions. *Proc. of the ACM SIGMOD*, 1990.

[5] A. K. Elmagarmid, Y. Leu, W. Litwin, and M. Rusinkiewicz. A Multidatabase Transaction Model for InterBase. *Proc. of the 16th VLDB*, 1990.

[6] MarketFeed The Telerate Consolidated Feed. *Technical Specifications*, August 1993.

[7] Telerate Ticker Feed Specification. *Technical Specifications*, August 1993.

[8] I. Greif and S. Sarin. Data Sharing in Group Work. *Computer-Supported Cooperative Work*, ed. Irene Greif, Morgan Kaufman Publishers, 1988.

[9] J. Pons and J. Vilarem. Mixed Concurrency Control: Dealing with Heterogeneity in Distributed Database Systems. *Proc. of the Fourteenth Conference on VLDB*, Los Angeles, 1988.

[10] Calton Pu. Superdatabases for Composition of Heterogeneous Databases. *Proc. of the Fourth International Conference on Data Engineering*, Los Angeles, 1988.

[11] Tore Risch. Monitoring Database Objects. *Proc. of the Fifteenth Conference on VLDB*, 1989.

[12] A. Rosenthal, S. Chakravarthy, B. Blaustein, and J. Blakeley. Situation Monitoring for Active Databases. *Proc. of the 14th VLDB*, 1989.

[13] Yuri Breibart, Hector Garcia-Molina and Avi Silberschatz. Overview of Multidatabase Transaction Management. *Stanford Technical Report No. STAN-CS-92-143*, May 1992.

[14] Michael Heytens, Sheralyn Listgarten, Marie-Anne Neimat and Kevin Wilkinson. Smallbase: A Main-Memory DBMS for High-Performance Applications. *Unpublished Draft* , October 1993.

[15] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On Rules, Procedures, Caching and Views in Database Systems. *Proc. of the ACM SIGMOD*, 1990.