A High-Frequency Data Repository for Financial Time Series

DAVID A BECK, DEVON E BOWEN, CURTIS S MEISSNER
OLSEN DATA AG, ZÜRICH SWITZERLAND
CONTACT: DATA-INFO@OLSEN.CH

Olsen Data AG
Seefeldstrasse 233, CH-8008, Zürich, Switzerland

26 March 1998
Current Version: 29 March 2001

Abstract

The relatively new field of high-frequency financial data analysis has significantly stressed current time series database implementations. Requesting tens of millions of irregularly spaced data points is a daily activity for researchers in the field. In addition, providing the ability to easily and accurately describe the data they need makes it possible for them to focus more on the research and less on the database implementation. In this paper, we present the data repository developed at Olsen Data to address these issues.

1 Introduction And Motivation

In 1985, the Olsen Group set out to collect high frequency financial data. Some financial instruments[1] are quoted at a frequency of a few times per day while others may be quoted a few times per second. This data has been collected around the clock for the last twelve years and time series containing tens of millions of prices have been collected for a number of instruments.

Our researchers use this data to look for correlations and phenomena in the financial markets on various time scales. We concentrate on examining the fractal nature of the data. For this type of research, it is essential to have access to all time scales. We, therefore, require that all data events (called *ticks*) be saved and that they be given a real timestamp. This leads to irregularly spaced datasets.

Fractal research is statistical in nature and, as such, often requires analysis of very large datasets for an instrument. Because keeping such large datasets in memory is not feasible without supercomputer technology, we have developed a data-flow-based, statistical package called ORLA for this purpose. ORLA acts like an electronic circuit in which a network of various off-the-shelf pieces is constructed and data flows through it to calculate a desired set of results (moving averages, trading model signals, etc). This eliminates the need for a large local memory, but this calculation model also requires a mechanism for slowly feeding data into the waiting ORLA process.

Finally, our researchers often need to request datasets that may not seem natural or obvious at first. For example, it may be desirable to request all currency exchange quotes for only Asian currencies, and again for European currencies, in order to compare the statistical distributions. Or we may request all quotes for all swap rates made by a given bank to see if that bank may be posting bad prices. It would be impossible to predict the scope of all possible data requests and to store the data appropriately from the start. So we wanted the database to be able to take a data description and return the desired time series.

After investigating all that the commercial database vendors had to offer, we found that none suited our needs. We then consulted the literature and found that surprisingly little research work has been done in this area. Most time series databases are geared toward small sets of regularly spaced data points. They usually assume a user wants an entire set at once. And they require the user to predefine these sets so that special data requests often are either not possible or are not easy. Because these systems fell far short of our needs, we felt the only solution was to develop our own.

While our requirements may sound specialized, we expect the need for database systems that can handle situations like this to grow. Research with high-frequency financial data is finding applications in diverse fields such as risk management and trading model development. Banks are embracing new solutions to these problems and often high-frequency data is being applied. In cases of risk management, for example, large matrices involving simultaneous access to thousands of high frequency time series need to be calculated.

[1]The word “instrument” is used to describe a specific financial contract such as a stock, currency exchange rate, or pork belly future.

Series	Time	Ccy	Price	Bank
1 2	13:00:00	CHF	1.4817	SEBM
1	13:00:02	JPY	130.64	PRUN
1	13:00:09	JPY	130.60	CHFX
1 2 3	13:00:10	DEM	1.8225	HYPO
1 2 3	13:00:12	DEM	1.8231	NWNC
1 2 3	13:00:16	DEM	1.8228	BOMX
1 2	13:00:19	CHF	1.4818	NWND
1 2 3 4	13:00:24	DEM	1.8226	BGFX
1 4	13:00:24	JPY	130.58	BGFX
1 2 3	13:00:24	DEM	1.8230	KOCT
1 2 4	13:00:24	CHF	1.4822	BGFX
1 2 3	13:00:30	DEM	1.8225	KBXE
1	13:00:31	JPY	130.58	DRE1

Figure 1: Example of four possible time series subsets of a larger time series. They are: 1) currency prices, 2) European currency prices, 3) German Mark prices, and 4) prices from the bank BGFX.

All this imposes stringent requirements on the financial database supporting the system. However, any new software product should be designed with generality in mind. For this reason, we made sure during development and implementation that our system was not limited to financial data. It may be used anywhere that high volume time series data needs to handled. Although significant effort was spent on optimizing and solving special financial world problems.

We have called our system a repository rather than a database because the word database tends to carry with it an assumption that data can be manipulated, joined, etc. What we needed was to store and retrieve large numbers of ticks and to do so flexibly. The word repository seemed a more accurate description of this functionality.

2 Time Series Model

A time series is a set of data points sorted in order of increasing time. In an abstract sense, one can define a “universal” time series as the time series of all recordable events that ever have and ever will occur. All other time series can be viewed as a subset of, or a restriction on, this universal set. Given any time series, a new time series can always be created by simply extracting a subset from it.

Figure 1 contains a list of currency prices over a 31 second interval. The currencies are the Swiss Franc (CHF), German Mark (DEM), and Japanese Yen (JPY). The first thing we need to note is that this list is already a subset of larger sets. Examples of supersets might include the time series of all currency prices or even of all financial price quotes for all instruments. Conversely, we can also break this series into subsets. We might ask for all European currencies from the set, or we may want only German Mark prices, or we may want only prices from the bank BGFX.

This is not just philosophical musing; all of these subsets have been of interest to our researchers at one time or another. And thinking about them in terms of restrictions on a superset is instructive because it can lead us to a model for data storage and, hence, to a language for repository query. Our goal was to build a repository which treated data in this way.

It should be noted that this model for time series data is not the usual one. Most databases require the user to prepackage the data they are storing into various files. For example, you might decide that you want the Swiss Franc currencies in one file and the German Mark in another and you would be required to predefine these files and to separate the data before storing it.

```
Tick        = ( Time , Item )
Item        = "FT" ( Contract , DataSpecies )

Contract    = FX | Deposit
FX          = "FX" ( Per , Expr )
Deposit     = "Deposit" ( Currency, Period )

DataSpecies = Quote | TX
Quote       = "Quote" ( Bid , Ask , Bank , Source )
TX          = "TX" ( Price , Volume , Seller , Buyer , Source )

Per         = string[3]:f
Expr        = string[3]:f
Currency    = string[3]:f
Period      = string[3]:f
Bid         = float:v
Ask         = float:v
Price       = float:v
Volume      = integer:v
Seller      = string:v
Buyer       = string:v
Bank        = string:v
Source      = string:f
```

Figure 2: A sample SQDADL definition to support currency exchage and deposit rates.

We felt that this preclassification was unnecessarily restrictive for our needs, required the user to have too much knowledge of the packaging method, and led to complicated query languages. For example, to get the BGFX bank quotes in our example above, the user would need to know that all these currencies are in separate files and would need to build a query by first combining these files and then asking for the BGFX quotes. This is clearly more complicated that simply asking for the BGFX quotes as a restriction of all known data.

3 Data Representation

The elimination of the file-based conceptual view means that each tick stands on its own in the repository without classification. For this to be useful, the data needs to be self-describing. This description can then take the place of the file as a handle for data queries.

We have developed a description language for this purpose which we call the Sequential Data Description Language (abbreviated SQDADL, and pronounced “skedaddle”). SQDADL is a BNF-style language with some restrictions to enforce a specific structure. Figure 2 presents a sample SQDADL description for the storage of either currency prices or interest rates (deposits).

By the definition of a time series, each tick must contain a timestamp and this fact is reflected in the root-level statement “Tick = (Time,Item)”, which forces all ticks into this form. This statement is required, although it is the only restriction placed on the data description. The “Item” reference can then be expanded as the user sees fit for the type of data to be stored. There is no implicit assumption that limits the repository to financial data.

Figure 3 shows the derivation of a description for a quote on a currency exchange. In this case, the FT indicates a “financial tick” (as opposed to some other time series data), the FX indicates “foreign exchange” from one currency to another, and the Quote indicates at what prices the given bank is willing to buy (Bid) and sell (Ask) one currency for another. In simple terms, the bank of CHFX is willing to sell Japanese yen at a price of 124.1 yen per US dollar and we were told this by the Reuters news agency.

This string contains all the information needed to allow this tick to stand on its own. If this string were

```
(Time,Item)
(08.02.1998 07:44:58,FT(Contract,DataSpecies))
(08.02.1998 07:44:58,FT(FX(Per,Expr),Quote(Bid,Ask,Institution,Source)))
(08.02.1998 07:44:58,FT(FX(USD,JPY),Quote(124.05,124.1,CHFX,REUTERS)))

(Time,Item)
(08.02.1998 07:49:34,FT(Contract,DataSpecies))
(08.02.1998 07:49:34,FT(FX(Per,Expr),TX(Price,Volume,Seller,Buyer,Source)))
(08.02.1998 07:49:34,FT(FX(USD,JPY),TX(124.1,1000000,CHFX,BGFX,REUTERS)))

(Time,Item)
(08.02.1998 04:51:47,FT(Contract,DataSpecies))
(08.02.1998 04:51:47,FT(Deposit(Ccy,Period),Quote(Bid,Ask,Institution,Source)))
(08.02.1998 04:51:47,FT(Deposit(USD,03M),Quote(5.5,5.62,BSBB,REUTERS)))
```

Figure 3: Examples of the parsing of some sample ticks. They are: 1) a foreign exchange quote, 2) a foreign exchange transaction, and 3) a cash deposit interest rate quote.

found written on a piece of paper on the floor, we would be able to enter it into the repository and then retrieve it as part of future queries. And yet we have not forced the user to separate its components into file and record specifiers. The only restriction is that it conform to the syntax of the SQDADL description file.

Figure 3 also illustrates how you can derive ticks for actual transactions and for interest rate deposit quotes. Given these definitions, interest rate deposit transactions also become possible. This is one of the nice features of the SQDADL language. Once the expansions of “Contract” and “DataSpecies” have been defined, they can be put together into various combinations which allows you to store many more instruments than you’ve even considered. The recursive nature of the language is also a significant win in the financial world because many contracts are, in fact, recursive. Relatively “simple” contract types such as options, futures, and bonds may be combined to create, for example, an option on a bond future contract.

There is also significant advantage in keeping the ticks in the form of strings. It allows parsing to be dynamic which means no code needs to be recompiled to handle new data types. You simply modify the SQDADL definition and you are immediately able to store ticks of the new type in the repository.

4 Time Series Request Syntax

Because each time series is modeled as a restriction of another time series, it is easy to see how the SQDADL definition can lead to a way of specifying queries to the data repository. We simply need to provide a syntax for restricting each of the fundamental types. The user can then combine these restrictions to define the desired time series. Restrictions are implemented with expressions.

4.1 General Expressions

Referring back to Figure 2, we notice that each of the “leaf nodes” of the SQDADL parse tree is given a type indicator. These types are known to the repository as fundamental types and closely follow types inherent in most programming languages or communications standards. For each of these types we define a set of expressions which can be used as a filter for deciding whether data is part of the requested series or not. This concept is very much like a regular expression or wildcard. In fact, for the string types, POSIX-style regular expressions could be directly used.

Thinking along these lines, we could send the following request to the data repository:

```
1. (*-*,FT(FX(USD,*),Quote(*,*,*,*)))
2. (*-*,FT(FX(USD,DEM|CHF),Quote(*,*,*,*)))
3. (*-*,FT(FX(USD,DEM),Quote(*,*,*,*)))
4. (*-*,FT(FX(USD,*),Quote(*,*,BGFX,*)))
```

Figure 4: SQDADL queries which select the corresponding time series as defined in Figure 1.

```
(*-*,FT(FX(USD,JPY),Quote(*,*,*,*)))
```

This request says that we would like all Japanese yen prices quoted against the US dollar over the entire range of time with no restriction on the prices, contributing bank, or the information source. Figure 4 provides examples of requests to match each of the time series previously defined in Figure 1.

It should be noted that there is a different expression syntax for each of the data types. For example, the syntax of an integer expression will be different than the syntax for a string. We determine which expression syntax will be used based on the types of the leaf nodes as indicated by the SQDADL parser.

It is not hard to imagine a set of expressions which allow the user to make very powerful and flexible filters for each data type. For example, one might use the expression "10 << 12" in an integer field to request only ticks with values between 10 and 12. These filters can be added and modified as time goes on since they only affect the data retrieved by a query and not the storage process.

In our implementation, we do, however, restrict expressions only to leaf nodes of the parse tree. That is to say that one cannot, in our example SQDADL, place a wildcard in a non-leaf node position such as:

```
(*-*,FT(*))
```

One might assume that a request like this would return all financial ticks whether they be currency ticks or interest rate deposits. This would be a logical assumption and while we found no reason that a syntax like this could not be implemented, it would be an unnecessary complication in our case. So it was left as an option for future enhancement.

4.2 Time Expressions

Because we are working with a time series repository, time is the handle by which we access our data. As such, expressions in the "Time" field are treated as a special case of the type-based expressions syntax.

To illustrate this, let's assume we want to get the price of an instrument as it was at midnight on a certain date. The probability of there being a tick at exactly midnight is actually very low so we usually need to ask for the tick before and the tick after so that some interpolation can be done. We might formulate this expression as "01.01.1990 00:00:00[-1..1]".

The problem here is that the time expression is no longer a filter whose behavior can be determined only by the tick itself. If we ask for the tick before midnight, we do not know if this tick will be one second, one minute, or even one day before that hour. The behavior of any filter that will include this tick depends not only on the time of the tick, but also on the temporal placement of other ticks in the specified time series.

This implies that the processing of the time expression is something that must be considered deep in the repository machinery since the low level features of the time series are only known here. While all other restrictive expressions can sit at a higher level, and even, theoretically, on the client side, time expressions must be handled specially.

5 Storage of Data Ticks

All modern operating systems support the concept of a file with an associated name and data. While our abstraction avoids the need for this classification of data on the user side, we must still at some point map our model onto the physical computer.

A trivial implementation of the storage and request system that we have described is to simply store all the strings that a user gives to the repository in a single text file. A request then only requires one to go through the file, apply the restrictions, and then return the specified subset. While this would work, it is quite inefficient. Obviously, we need to employ some kind of grouping of like data behind the scenes in order to improve data query performance.

What we have implemented is an architecture which allows the repository itself to store the data in the most efficient way it can based on hints given to it in the SQDADL configuration file. Referring back to Figure 2, you will notice that all leaf nodes in the parse tree not only have a type assigned to them but also a designation 'f' or 'v'. This value is a hint to the repository and indicates whether this field is considered fixed or variable with respect to the most common query for data.

For example, in the sample SQDADL configuration you will note that the currencies are all tagged with the "fixed" hint. This means that the we expect users to more often ask for a fixed currency in their requests rather than a broader expression as a filter. Specifically, we expect more queries of the form:

```
(*-*,FT(FX(USD,JPY),Quote(*,*,*,*)))
```

than:

```
(*-*,FT(FX(USD,*),Quote(*,*,*,*)))
```

With these hints, we have all that we need in order to store the data in a file on the physical machine. Given a string representation of a tick, we create two data buffers into which we will divide the string. The first will become the filename and the second will hold the data record which will be appended to this file.

Because we want to ensure that we have random access capabilities in each file, we need to guarantee that each data buffer is the same length for a given file. This, of course, depends on the type of the data going into the buffer. For types of fixed size, for example, integers, we simply write the data into the buffer. However, if the data is variable in size, such as a variable length string, we need another solution. In this case, for each file we create a secondary storage file to hold all variable length data and its size. The offset into this file is then stored in the data buffer. Since the offset is simply an integer, we maintain a fixed size for all records in the primary file.

Now we parse the tick and divide the fields into the two buffers according to the following straightforward rules:

- If the field is a non-leaf node token or it is a leaf node token with a hint of "fixed", copy it to the filename buffer.
- If the field is a leaf node with a hint of "variable" and with a constant size, copy it to the data buffer. Then place a "*" in the filename buffer.
- If the field is a leaf node with a hint of "variable" and has a non-constant size, write its size and data to the secondary file and copy its offset to the data buffer. Then place a "*" in the filename buffer.

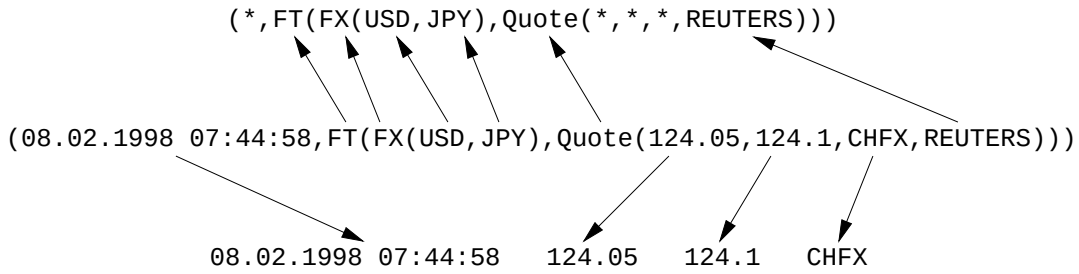

Figure 5: Separation of a fully described tick into its filename (top) and data record (bottom) components. Note that all data is recorded and nothing is implied.

Figure 5 shows how a given foreign exchange quote tick would be broken up into the two buffers assuming our earlier SQDADL configuration. For efficiency, the data record buffer holds binary versions of the data. And because the filename is often long and a rather strange collection of parentheses, wildcard characters, and commas, we are required, under most operating systems, to use a layer of indirection here in order to map the appropriate filename onto a filename that the operating system can handle natively.

Once the parsing is complete, we simply open the appropriate file and append the data buffer onto the end. If the file does not already exist, it is created beforehand. In this way, the repository is dynamic and can adapt to new ticks (for example, the creation of a new currency) but still hides the maintenance of this from the user.

6 Retrieval of Data Ticks

The hints given in the SQDADL definition lead to a pattern of possible filenames. For example, we can easily show that the hints we have given in Figure 2 could lead to the filename:

```
(*,FT(FX(USD,JPY),Quote(*,*,*,REUTERS)))
```

If the user submits this same string as a request for data (with some range of time) we can simply open the file, search for the appropriate start time, and give the ticks to the user until the end time is reached.

Of course, this is an optimal request. Our system also needs to be able to handle non-optimal requests and allow users to specify expressions anywhere they please without having to know how the data is actually stored.

6.1 File List Selection

Given a request for a time series, we need to be able to determine all possible files that may have information relevant to the request. This is a straightforward operation. We parse the request into tokens and then apply three rules to the set of all filenames:

- If the given token is a non-leaf node, then select filenames that exactly match it.
- If the given token is a leaf node and this leaf has a hint of “variable”, then select filenames that have a “*” in this position.
- If the given token is a leaf node and this leaf has a hint of “fixed”, then apply this expression and select only those filenames that match it.

Using an example from Figure 4, if we are given the request:

```
(*,FT(FX(USD,*),Quote(*,*,BGFX,*)))
```

and the following filenames exist in our repository directory:

```
(*,FT(FX(USD,JPY),Quote(*,*,*,REUTERS)))
(*,FT(FX(USD,CHF),Quote(*,*,*,REUTERS)))
(*,FT(FX(USD,DEM),Quote(*,*,*,REUTERS)))
(*,FT(FX(DEM,CHF),Quote(*,*,*,REUTERS)))
(*,FT(FX(DEM,GBP),Quote(*,*,*,REUTERS)))
```

we would only select the following files for reading to service this request:

```
(*,FT(FX(USD,JPY),Quote(*,*,*,REUTERS)))
(*,FT(FX(USD,CHF),Quote(*,*,*,REUTERS)))
(*,FT(FX(USD,DEM),Quote(*,*,*,REUTERS)))
```

Given the hints in the SQDADL definition, it is clear that only these files can contain information that we are interested in. The expression "USD" prevents the selection of the others. Yet because it is a variable field, the "BGFX" expression does not affect the list.

6.2 The Data Cursor

Once we have established a list of possible files we need to create a software *cursor* which can be used to pass over the files and hand the ticks to the user when requested. In object oriented terms, we instantiate a cursor object by giving it a set of files from which to read, a desired starting time, and the entire expression pattern that defines the desired time series. Internally, it then opens each of these files and does a binary search to find the desired start time.

Once instantiated, the cursor provides two methods to the user, called *next()* and *prev()*. The next() method returns the nearest tick in the requested time series after the current time. The prev() method returns the nearest tick in the requested time series immediately before the current time.

How the cursor actually does this is displayed graphically in Figure 6. When asked for the nearest tick after the current time, it surveys all the files and chooses the tick with the lowest timestamp. It then applies the expression filter to this tick to see if it should be included in the requested time series. If not, it goes back to the files to get the next until a matching tick is found. Conceptually, this is merging the files in time series order and then removing any ticks that are not appropriate. The prev() method has the same implementation, but works by backing up in the files rather than moving ahead.

6.3 Servicing a Request

It is now quite trivial to see how a wrapper can be made around this cursor to service any request for data. For example, let's say the user has given us the following request string:

```
(01.01.1990 00:00:00[-10..5],FT(FX(USD,*),Quote(*,*,BGFX,*)))
```

This means that we want all ticks for any currency measured against the US dollar that came from the bank BGFX. And our time range is the 10 ticks before midnight 01.01.1990 and 5 ticks after. Our steps for handling the request are then:

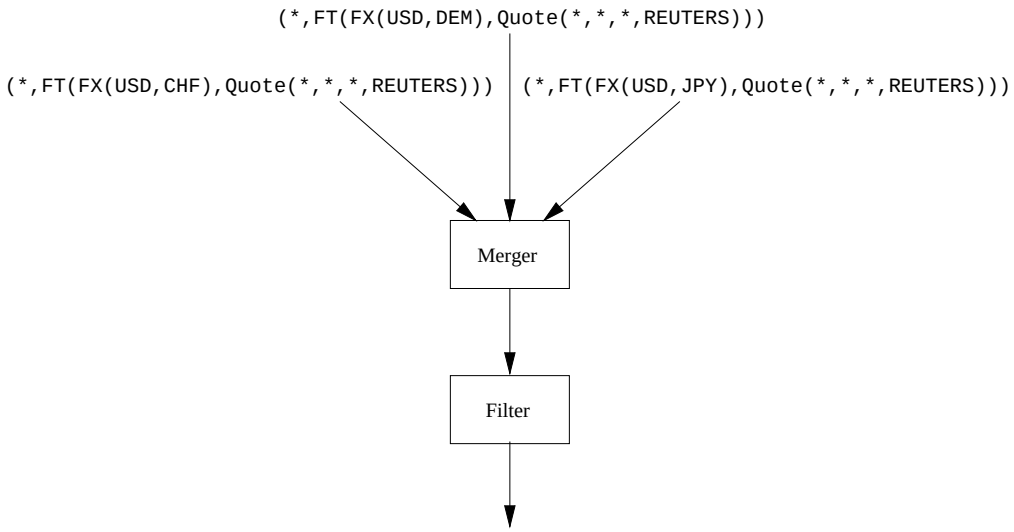

Figure 6: The data cursor is a software object which knows how to merge ticks from all the data files and remove all undesirable ticks. Valid ticks are then returned to the user. The cursor is built to work either forward or backward in time.

- Build the list of all possible filenames that could contain data we need for this request. This was done in the last section.
- Extract the base time from the time expression. In this case, our base time is "01.01.1990 00:00:00".
- Instantiate a cursor for these files with this start time and pass it the full expression pattern we were given.
- Make *n* calls to the prev() method to rewind our time series. In this case, *n* is 10.
- Make *n* calls to the next() method and hand each to the user. In this case, *n* is 15, and represents the total length of the time series.

Here, the merging of all files by the cursor provides us with all possible appropriate ticks. But the cursor tests against our full expression to ensure that only those from bank BGFX are actually given to us. This is the time series we requested.

7 Comments on Administration

It should be clear from this description that the expensive part of a request is data removal. Because a bank name has a hint of "variable", it is put inside the data record rather than in the filename. This means we will often need to read many data records that do not match our expression just to get at those that do. This is a waste of computer time.

One solution to this problem would be to give the bank name a "fixed" hint. If this were the case, then it would be put in the filename and we would only need to open and merge the files that we really need. This would result, again, in a near optimal request because the merge operation is computationally trivial.

The decision as to whether to make a leaf node "fixed" or "variable" needs to be made by the repository administrator. If it is known that there is a small number of banks, then making "Bank" a "fixed" field may be a reasonable option, since only a few additional files would be created. On the other hand, we

would certainly not want to define the price field as “fixed”, since there are essentially an infinite number of prices.

Another factor must be taken into account as well. The administrator needs to decide which are the most common fields to be fixed in a user request. If users rarely put restrictions on the bank field, then we are not hurting anyone significantly by putting this in the data record because computer time will only be occasionally wasted.

As we can see, it is not only the data itself that determines how the files are arranged but also the requests. When it is difficult to decide, we find that it is better to err on the side of making a leaf node “fixed”. This results in faster request processing because fewer ticks need to be removed at run time. However, it should be stressed that the storage hints in no way affect the requests that the user can make. They only affect how fast those requests are handled. Indeed, the administrator can reorganized the file storage unbeknownst to the user.

8 Experience

While our experience with the described design has been very positive thus far, there are a few problems that we have identified. These need to be addressed in future development before the system can be considered complete.

The SQDADL code was designed to be flexible and easy to maintain. The goal was that we should be able to add new data types to the repository in a matter of minutes simply by defining the syntax of a new type and specifying the breakdown of its fields. This has been achieved and we have been able to expand our data collection with very little effort given to making SQDADL definitions.

Modifications to the SQDADL file have to be made carefully, however. If a set of data files already exists assuming one SQDADL definition and then this definition is changed, unexpected results will occur. The administrator needs to be careful to ensure that only new string patterns are added and that none already existing are changed. If it becomes necessary to change a definition, a new definition must be made, the files then copied through the repository interfaces from the old to the new, and then the old removed. This is a painful process at the moment and should be made more automatic.

Because SQDADL fully describes the financial instrument, a complex instrument such as an option on a bond future is represented by a complex SQDADL syntax. This makes it difficult for end users to remember the syntax. There are two possible ways to handle this problem. First, a layer could be built on top of the normal repository requests so that simple data requests could be done simply, leaving more complex requests to be done through the normal syntax. Alternatively, a tool could be developed to help the user dynamically build the requests strings by listing options and filling in boiler-plate components as needed. We envision a functionality similar to the UNIX tcsh command interpreter or the X windows xfontsel font browsing utility.

Related to this is the lack of meta information about what is actually stored in the repository. For example, the user may know that currency prices are stored but is it possible to find the list of those that are available? In the current design, the user simply has to know. But a meta-query database should be made available to store the various possibilities for each leaf node of a request string. A user could then ask what currencies are available. And this could, again, be built into a simple tool.

As it stands now, a request for data that does not exist simply returns a null list of data. This is a natural result of the design because a request string is really saying “give me all the data that you have that matches this pattern”. This subset can, of course, be the null set. But this is often not what people expect as a return value. Some other way of returning an error should be developed.

Finally, the use of flat files for data storage requires that all data arrive in time order so that it may be

stored that way. This has not been a problem for us so far. However, if this becomes an issue, we could simply use a b-tree storage mechanism in the lower layer. This would add some overhead but would also solve all time ordering issues that could arise.

9 Conclusion

The described system has so far shown itself to be quite useful in the field. The flexibility of the SQDADL language has allowed us to collect over 30 instrument types with very little effort being spent on the data definition. The implementation has also resulted in fast response times because of the flat-file storage foundation. While there continue to be issues that need to be addressed, none seem unsolvable and progress is continuing.

10 Acknowledgements

We would like to thank Gilles Zumbach, whose ideas for splitting financial instruments into various contractual components led to the development of the SQDADL language for describing the data; Bill Kelly for his help in cleaning up details of the language and for his frequent reviews of our concepts; Paul Breslaw who, as our first user, pointed out a number of issues to be addressed; and Giuseppe Ballocchi and Vera Haas for their help in reviewing this manuscript.

11 Bibliography

Chandra R, Segev A: *Managing Temporal Financial Data in an Extensible Database*, Proceedings of the 19th VLDB Conference, Dublin, 1993.

Dreyer W, Kotz Dittrich A, Schmidt D: *Research Perspectives for Time Series Management Systems*, SIGMOD RECORD, Vol. 23, No. 1, March 1994

FAME Software Corporation: *User's Guide to FAME*, 1990.

Informix Corporation: *Informix TimeSeries DataBlade Module*,
http://www.informix.com/informix/bussol/iusdb/databld/dbtech/sheets/tmsrdb.htm.

Sadamichi H: *On SQL and its application to the Time Series Database*, Kokumin Keizai Zasshi, Vol. 164, No. 5, November 1991.

Schmidt D, Kotz Dittrich A, Dreyer W: *Time Series, a Neglected Issue in Temporal Database Research?*, UBILAB Conference Proceedings, Zürich, 1996

Shaw J, Ward R, Zumbach G: *The Book of ORLA*, Internal Document, the Olsen Group, Zürich, June 1996.

Tansel A, et al: *Temporal Databases, Theory, Design, and Implementation*, Benjamin Cummings, 1993