Pipelining Vector-Based Statistical Functions for In-Memory Analytics

A New Technique to Reduce Latency in Managing Market Data

McObject LLC
33309 1st Way South
Suite A-208
Federal Way, WA 98003
Phone: 425-888-8505
E-mail: info@mcobject.com
http://www.mcobject.com

Introduction

Managing trade- and quote-related market data is a key task for software underlying today's automated capital markets, including applications for algorithmic trading, risk management and order matching and execution. Reducing latency in these systems can deliver a competitive advantage, so technology that accelerates market data management is eagerly welcomed.

Market data typically takes the form of a time series, or repeated measurements of some value over time. Database management systems (DBMSs) have evolved specialized techniques to speed up time series processing, including column-based handling of such data. This paper examines the column-based approach as implemented by McObject's *e*X*treme*DB® Financial Edition database system. It looks at how *e*X*treme*DB Financial Edition implements columns, with database designs that support hybrid column- *and* row-based data handling – and focuses on a key, related *e*X*treme*DB Financial Edition feature for reducing latency: its library of vector-based statistical functions, which are designed to execute over data sequences (columns). The paper shows how columnar storage improves performance by maximizing the proportion of relevant market data that is loaded into CPU cache, and how pipelining of vector-based statistical functions maximizes performance by eliminating costly transfers between CPU cache and DRAM.

Columnar Data & the "Sequence" Data Type

Relational DBMSs use the concept of tables, consisting of rows and columns, to logically arrange data. Object-oriented DBMSs, and "object-friendly" database systems like *e*X*treme*DB Financial Edition, replace tables with classes of objects – but conceptually, classes and tables are the same, and for simplicity, this paper will refer to "tables."

Traditional (both relational and object-oriented) DBMSs are row-oriented, in that they act on stored data in a row-by-row fashion. Database systems organize rows into a database *page* that is the basic unit of input/output for a DBMS. So, when data is read by the DBMS, whether from the database's cache or persistent media, a page is transferred into the CPU cache. For example, given the table shown in Figure 1, below, and a database page size of 4096 bytes, a page would hold about 145 rows of data. The figure illustrates two database pages. The row number is shown strictly for illustrative purposes.

Row#	Symbol	Open	Close	Volume	Date
1	IBM	204	205	200000	20120410
2	IBM	202	203	150000	20120411
3	IBM	204	205	300000	20120412
…	…	…	…	…	…
145	IBM	206	202	400000	20120927
150	IBM	202	203	200000	20120928
151	IBM	203	202	200000	20120929
152	IBM	202	205	400000	20120930
…	…	…	…	…	…
290	IBM	210	209	150000	20130316

Figure 1.

The problem with the row-based approach is that for time series analysis (and hence market data analysis), only a sub-set of the columns is likely to be of interest for a given operation. For example, many financial calculations would use only closing prices (the Close column in Figure 1). Row-oriented processing results in entire pages being fetched into CPU cache, including irrelevant Symbol, Open, Volume and Date elements. In other words, significant bandwidth is wasted because only a quarter of the data fetched is actually used. The Close column is used in the calculation, but the Symbol, Open, Volume and Date columns are transferred to CPU cache only because of the way data is organized in a conventional row-oriented DBMS.

This issue of “flooding” the cache with unwanted data is addressed by column-oriented database systems that – as their name implies – bring data into CPUcache column-by-column. In the example discussed above, but taking a column-wise approach, a column of Close data is collected into a database page, up to the page size limit. When this page is moved into CPU cache, much more of the Close column data is transferred to CPU cache per transfer, compared to using the row-based approach. The transfer bandwidth is used more efficiently, far fewer fetches are required, and this reduces latency.

However, while column-based handling accelerates analytical processing on columns of data, row-based management is generally faster elsewhere, i.e. for “general” data processing that requires operations on multiple columns. In fact, many capital markets applications manage data that is naturally columnar alongside data that is best-served using a row-based approach.

*eXtreme*DB Financial Edition gives developers the flexibility to implement column-based *and* row-based data handling and use a hybrid approach that combines the two models in a table. It accomplishes this through its unique “sequence” data type. A C/C++ developer working with *eXtreme*DB Financial Edition creates a database schema (design) in a text file and compiles it using McObject’s *mcocomp* utility, resulting in the header files and database dictionary to be used in applications with that database. The schema declares data types for the different columns in a table. For example, the schema below creates the Security table (class) with one column consisting of fixed-length character data, five columns of variable-length string data, and nine columns declared to be of the sequence data type:

```
class Security
{
   char<16> Id;
   string  ID;
   string  Ex;
   string  Descr;
   string  SIC;
   string  Cu;

   sequence<time asc> TradeTimeStamp;
   sequence<float> TradePrice;
   sequence<uint4> TradeSize;

   sequence<time asc> DateStamp;
   sequence<float> Open;
   sequence<float> Close;
   sequence<uint4> Volume;

   sequence<time asc> AskTimeStamp;
   sequence<float> AskPrice;
   sequence<uint4> AskSize;

   sequence<time asc> BidTimeStamp;
   sequence<float> BidPrice;
   sequence<uint4> BidSize;
};
```

In the table above, char and string are scalar data types, which can hold a single element. In contrast, a sequence is an unbounded array, and can hold multiple elements. Two or more sequences can be considered conceptually as a time series. In the example above, four groups of sequences can be considered four time series. Certain programming languages enable operations on vectors, but generally require that the vectors reside in memory, which limits their usefulness. (If the vectors to be operated on are very large and/or the available system memory is too small, the system will either be unable to complete the operation or performance will degrade because the operating system has to swap the data in and out of physical and virtual memory.) Further, with a vector-based language an expression such as

```
result = a * b / c
```

the interim result of the operation `a * b` (another vector) must be produced (materialized) before the next operation (division of that vector by the vector `c`) can be executed. If, for example, a and b are vectors of 100 million elements, each consisting of a 4-byte unsigned integer (400 million bytes), then a 3rd interim vector of 400 million elements must be produced for the interim result (so that each element can be divided by the vector elements of c to produce the final result vector). Naturally, a 400 million byte vector does not fit in the CPU's cache, so it has to be created in memory (DRAM), requiring many costly, latency-inducing moves between the CPU and DRAM.

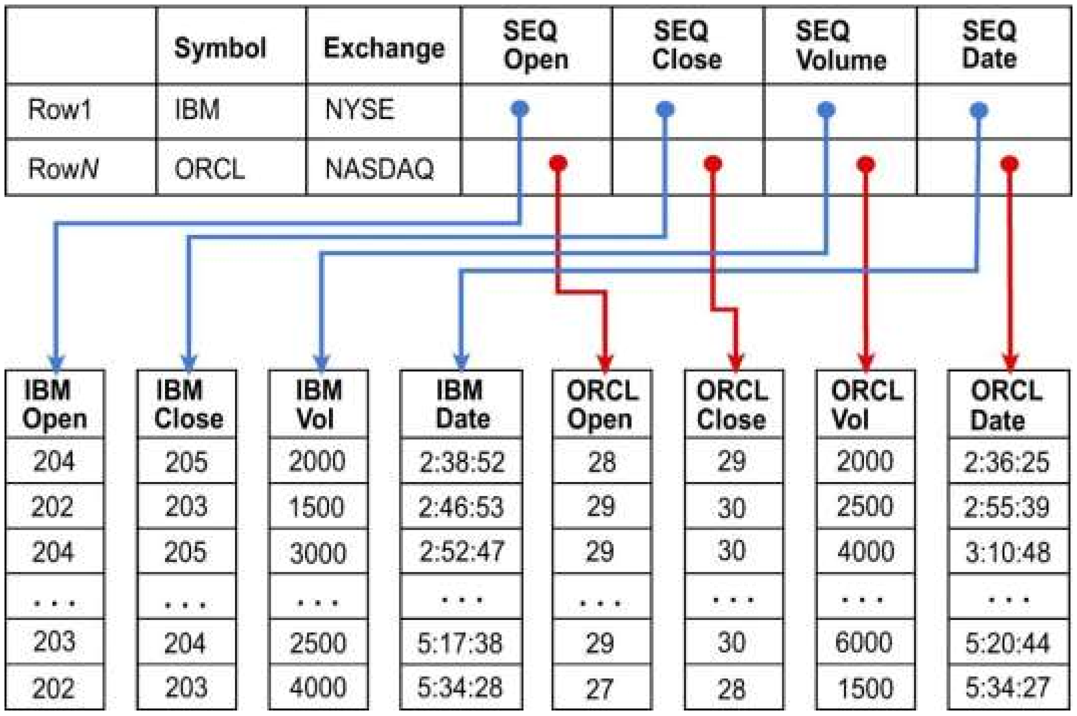
*e*X*treme*DB Financial Edition is written in the C programming language, and *e*X*treme*DB users generally program in C or a similar language (C++, Java, C#, Python). None of these languages are vector programming languages, but the C language can be used to implement vector (array) processing, and *eXtreme*DB Financial Edition provides a library of more than 150 functions for vector-based statistical analysis of sequences.

In order to overcome vector programming languages' restriction that entire vectors reside in memory, *eXtreme*DB Financial Edition implements a specialized type of database page called a *tile* to fetch sequence data into CPU cache. McObject uses the term *tile-based processing of vector elements* to describe *eXtreme*DB Financial Edition's handling of time series data, using the product's vector-based statistical functions to work with data stored as sequences. The effect is that – given the table with Open, Close, Volume and Date elements presented above – only the tiles of Close data needed for processing are brought into CPU cache, as shown in Figure 2 below.

Close	204	202	204	204	202	204
203	204	205	206	208	201	205
204	203	202	200	201	205	204
203	202	203	201	205	207	206

Figure 2.

Fields defined as sequences in *eXtreme*DB Financial Edition reside in tables alongside all other supported data types; the database system "knows" to handle the time-series data in a column-wise fashion, as shown in Figure 3 below, and to apply row-based processing elsewhere.



	Symbol	Exchange	SEQ Open	SEQ Close	SEQ Volume	SEQ Date
Row1	IBM	NYSE	●	●	●	●
Row*N*	ORCL	NASDAQ	●	●	●	●

IBM Open	IBM Close	IBM Vol	IBM Date	ORCL Open	ORCL Close	ORCL Vol	ORCL Date
204	205	2000	2:38:52	28	29	2000	2:36:25
202	203	1500	2:46:53	29	30	2500	2:55:39
204	205	3000	2:52:47	29	30	4000	3:10:48
. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .
203	204	2500	5:17:38	29	30	6000	5:20:44
202	203	4000	5:34:28	27	28	1500	5:34:27

Figure 3.

Pipelining Vector-Based Statistical Functions

Pipelining vector-based statistical functions is a key *eXtreme*DB Financial Edition programming technique to accelerate performance when working with time series data, such as market data. One powerful effect of pipelining is that it causes intermediate result sets used in processing to remain in CPU cache, rather than being output as temporary tables, as would be required with other database systems, including column-oriented DBMSs and vector/array languages. Keeping intermediate results in CPU cache eliminates the latency imposed by back-and-forth trips across the Quick Path Interconnect (QPI) or Front Side Bus (FSB) between CPU cache and main memory.

Here's how pipelining works, with examples using *eXtreme*DB Financial Edition's vector-based statistical functions in C and in the SQL database programming language. Let's say the application needs to calculate the 5-day and 21-day moving averages for a stock and detect the points where the faster moving average (5-day) crosses over or under the slower moving average (21-day). This might be desired by traders who view a cross over as a market entry point (a buy or short signal), and a cross under as an exit point (a sell or cover signal).

This processing is accomplished in the C code below, using four functions from the *eXtreme*DB Financial Edition vector-based statistical function library, and a sequence of historical closing prices as input:

```
mco_seq_window_agg_avg_double(&5day, &close, 5);
mco_seq_window_agg_avg_double(&21day, &close, 21);
mco_seq_sub_double(&delta, &5day, &21day);
mco_seq_cross_double(&crosspoints, &delta, 1));
mco_seq_map_double(&crossprices, &crosspoints, &close));
```

1. The first function, 'mco_seq_window_agg_avg_double', executes over the input sequence of closing prices referenced by '&close' to generate the average of elements within a given window (in this case, the window is five days). Its result – the sequence of 5-day moving averages – is referenced by a special kind of data object called an *iterator*. An iterator provides a pointer (a reference) to an element within a sequence, and the iterator created here, '&5day', refers to 5-day moving averages.

   Note that while the '&5day' iterator becomes input for subsequent statistical functions in this code example – and is treated as if it were a sequence within the database – it is never actually materialized as output and transferred into main memory or storage. Instead it "lives" in CPU cache at the time of program execution.

2. The same function, 'mco_seq_window_agg_avg_double', executes over '&close' and returns '&21day', an iterator referencing 21-day moving averages.

3. The vector-based function 'mco_seq_sub_double' executes over the iterators that have been created, subtracting each element of '&21day' from the corresponding element of '&5day', resulting in a new iterator '&delta' that, as its name implies, is the delta between the 5-day and 21-day moving averages.

4. The function ‘mco_seq_cross_double’ takes the iterator ‘&delta’ as input and finds the crossover points by detecting the points (element positions) at which ‘&delta’ elements cross zero, i.e. go from positive to negative or vice versa. This is returned as ‘&crosspoints’.

5. The final function, ‘mco_seq_map_double’, maps the crosspoints to the original sequence of closing prices. In other words, it returns a sequence of closing prices at which the 5-day and 21-day moving averages crossed.

The SQL code below accomplishes the same thing, using *e*X*treme*DB Financial Edition’s vector-based statistical functions as arguments to a SELECT statement:

```
select seq_map(ClosePrice,
seq_cross(seq_sub(seq_window_agg_avg(ClosePrice, 5),
seq_window_agg_avg(ClosePrice, 21)), 1))
from Security;
```

1. Two invocations of ‘seq_window_agg_avg’ execute over the closing price sequence, ‘ClosePrice’, to obtain 5-day and 21-day moving averages. Note that in SQL, the iterators are internal to the SQL engine and are not referenced in the code.

2. The function ‘seq_sub’ subtracts 21- from 5-day moving averages;

3. The result “feeds” a fourth function, ‘seq_cross’, to identify where the 5- and 21-day moving averages cross.

4. Finally, the function ‘seq_map’ maps the crossovers to the original ‘ClosePrice’ sequence, returning closing prices where the moving averages crossed.

In both the SQL and C code examples above, performance is accelerated by bringing only closing prices into CPU cache for processing at the start of the operation, even if the data comes from a table with multiple columns, as in Figure 3 above. Perhaps even more significant – in terms of minimizing back-and-forth trips between CPU cache and memory – is that this approach eliminates the need to create, populate and query temporary tables outside CPU cache in main memory, as would be required to manage intermediate results of processing with other database systems and vector-based programming languages.

This is achieved because of eXtremeDB’s *tile-based processing of vector elements*. Specifically, one tile of input data is processed by each invocation of ‘mco_seq_window_agg_avg_double()’, each time producing one tile of output that is passed to ‘mco_seq_sub_double()’ which, in turn, produces one tile of output that is passed as input to mco_seq_cross_double(), which produces one tile of output that is passed to mco_seq_map_double(). When the last function, mco_seq_map_double() has exhausted its input, the whole process repeats from the top to produce new tiles for additional processing.

In contrast, to accomplish the task discussed above (finding the crossover points of 5-day and 21-day moving averages) using a traditional SQL DBMS, the developer first creates three temporary tables:

```
CREATE TEMP TABLE mavg ( 5day float, 21day float );
CREATE TEMP TABLE sub ( delta float );
CREATE TEMP TABLE crosses ( Price float );
```

The next step calculates 5-day and 21 moving averages and populates the table ‘mavg’ with the results (this code assumes that the database system supports user-defined functions):

```
INSERT INTO mavg SELECT MovingAvg( ClosePrice, 5 ), MovingAvg ( ClosePrice,
21 ) FROM Security;
```

The next step populates the temp table ‘sub’ with the result of subtracting the two moving averages:

INSERT INTO sub SELECT 5day – 21day FROM mavg;

From here, “normal” SQL can go no further; the developer must write some code (shown here in C) to get the crossover positions:

```
// create two arrays to hold the positions of crosses over/under zero
int overs[100];
int unders[100];
// indexes into the two arrays above
int j = 0, k = 0;
rc = SQLExecDirect( “SELECT delta FROM sub” );
rc = FETCH FIRST into “PrevDelta”
for( pos = 0; rc == S_OKAY; pos++ ) {
   FETCH NEXT into “ThisDelta”
   if ThisDelta < PrevDelta AND ThisDelta < 0 AND PrevDelta > 0
      unders[j++] = pos;
   else if ThisDelta > PrevDelta and ThisDelta > 0 and PrevDelta < 0
      overs[k++] = pos;
   PrevDelta = ThisDelta
}
```

The next step fetches closing prices and inserts them into the temp table ‘crosses’ at the positions previously determined to be crossover points:

```
rc = SQLExecDirect( “SELECT ClosePrice from Security” );
int count = 0;
j = 0, k= 0;
while( rc == S_OKAY ) {
   if( count == overs[j] {
      INSERT INTO crosses VALUES ( ClosePrice );
      j++;
   } else if( count == unders[k] ) {
      INSERT INTO crosses VALUES (ClosePrice );
      k++;
   } else {
      ; // do nothing, this ClosePrice was not a cross over or under
   }
   count++;
}
```

Now the temp table ‘crosses’ has the closing prices where the 5-day and 21-day moving averages crossed (but not in which direction: over or under zero. Calculating that information is a “bonus” feature of the *eXtreme*DB Financial Edition SQL code presented above; to accomplish this using the traditional SQL DBMS would require additional code).

Clearly, using the traditional SQL DBMS requires a lot more code. But more importantly, all temporary (transient) results used in processing have to be ‘materialized’ or created as output that is external to CPU cache. In the traditional SQL approach, the first SELECT statement queries the database's closing prices table (‘ClosePrice’) to populate a new, temporary table. Closing price data has to be assembled into a page or pages and brought "across the transom" - through the Quickpath Interconnect (QPI) or front side bus (FSB) separating CPU cache from RAM - from the ClosePrice column in database storage into CPU cache for processing, then written back (across the transom in the other direction) to the temporary table. This is repeated to populate the second and third temporary tables, and finally, the third temporary table has to be read again, with results brought back from RAM into CPU cache, and inspected for the crossover points.

Note that a columnar SQL database would gain some efficiency by eliminating the waste inherent in row-based layout. But without pipelining, the overhead due to traffic back and forth across the QPI or FSB, which is several times slower than CPU cache, is enormous. The delay generated by these transfers is multiplied by the number of pages required to move a given set of data back and forth. In other words, processing by the SQL DBMS wouldn’t be just 3-4 times slower than the approach using pipelining. It would 3-4 times slower *times* the number of pages that have to travel across the QPI/FSB, whereas pipelining reduces the number of transfers for interim results to zero.

Performance & *eXtreme*DB Financial Edition

What kind of performance do *eXtreme*DB Financial Edition’s ‘sequence’ data type and pipelining of vector-based statistical functions deliver, when put to the test? McObject and partners Kove and Fultech Consulting recently implemented a risk management proof-of-concept (POC) to address a growing demand for comprehensive risk analysis on a compressed timescale. The POC measured exposure of 50,000 portfolios, each with 3,278 positions, using a 10-year historical value-at-risk (VaR) model. Target processing time was set at 10 minutes, reflecting a growing demand in this application category for intraday and even real-time risk analysis.

In fact, *eXtreme*DB Financial Edition accomplished this massive crunching of market data in *less than 8.5 minutes*. It performed approximately 428 billion calculations on approximately 13.5 terabytes of raw market data, reduced to 3.5 terabytes of Kove® XPD™ L2 storage through database normalization (host servers consisted of four 4-socket socket Dell R910s, with each socket housing an 8-core CPU, for a total of 128 cores. Each server had 1024 GB of RAM). This result equates to continuous financial model processing of approximately 950 million calculations per second and 65 terabytes per hour (learn more about the POC).

A new regulatory environment – specifically, mandates emerging under Dodd Frank, Basel III and other reforms for trading organizations to more closely monitor counterparty risk – provides the context for this proof-of-concept. But competitive and technical challenges in capital markets are also driving the critical need for improved real-time database solutions. Market data volume is soaring, with the NYSE producing more than a half billion trades and quotes on a typical day, and exceeding two billion on peak days. Capital markets technology must digest, sort and analyze *more* data – and do more with it, with increasingly sophisticated applications and algorithms. A DBMS for real-time finance systems must also leverage increasingly powerful multi-core hardware with high-performance networking.

In short, the DBMS must not become a processing bottleneck, and this goal is met with a variety of new approaches, including in-memory storage, column-oriented layout of market data, and CPU cache-optimized statistical analysis functions. *eXtreme*DB Financial Edition incorporates these advances: it is an in-memory database system (with optional persistent storage) and – as discussed in detail above - offers columnar handling that can be applied to time series data, while retaining more efficient row-based processing elsewhere.

Importantly, *eXtreme*DB Financial Edition is a *complete* database system, with transactions supporting the ACID (Atomicity, Consistency, Isolation and Durability) properties to safeguard data integrity, a database definition language, multi-threaded concurrent access, database indexes (B-tree, KD-tree, Hash and more), built-in clustering and high availability, industry standard SQL, ODBC and JDBC interfaces as well as native C, C++, Java,.NET and Python APIs, and a track record of 13+ years and tens of millions of successful deployments in demanding applications.

Finally, *eXtreme*DB Financial Edition offers some truly unique performance-enhancing characteristics to differentiate capital markets applications. One is an ultra-short execution path, with a code “footprint” of approximately 150K. This short code path speeds execution by reducing the number of CPU cycles required per database operation and by increasing the likelihood that instructions needed for an operation is already in CPU cache. Another unique and powerful feature is the pipelining discussed in this paper. This programming technique builds on *eXtreme*DB Financial Edition’s column-based handling of market data, accelerating processing by ensuring sequences of market data are exactly where they’re needed: in CPU cache. Pipelining represents a new approach to minimizing latency, for financial systems on the frontier of technology-based competitive advantage.