Reprinted with permission:

TABB FORUM
WHERE CAPITAL MARKETS SPEAK
Spotlight: The Bionic Trading Desk

By Steven Graves, McObject CEO and co-founder
Published on November 5, 2015

Is SQL Fast Enough for Tick Data?

Most enterprise systems query, sort and analyze data via database management systems (DBMSs) based on the SQL database programming language. Historically, tick data management in capital markets has stood out as an exception: Trading systems have eschewed SQL due to its perceived performance latency and unpredictability, in favor of more labor-intensive, lower-level programming languages (namely q and C/C++) for database operations. While this approach yields fast systems, disadvantages include slower time to deployment compared to using SQL – a curse for financial technology, where a technical advantage can be very profitable, but often only for a limited time. It also results in a mismatch between the skills that trading technology shops need (low-level languages) vs. a labor pool heavily slanted toward SQL skills and experience.

But is SQL really too slow for trading systems? Judging from the results of recent benchmark tests, that idea might be as outdated as the image of buyers and sellers gathered in trading pits, giving hand signals, shouting orders and scribbling transactions in notebooks.

If there is a gold standard for judging tick data management performance, it is the STAC-M3 benchmark specified by the Securities Technology Analysis Center (STAC). STAC develops and publishes numerous benchmarks; it closely monitors vendors' implementations of these tests, and audits results, so that its published reports provide true apples-to-apples comparisons of the technology stacks (hardware and software) used. Notably, while vendors provide the equipment, developers and testing fees for rounds of benchmark testing, it is STAC itself that publishes and stands behind the results.

The STAC-M3 benchmark focuses on tick analytics, measuring performance on complex queries that were designed by trading firms on the STAC Benchmark Council to reflect real-world capital markets computing demands. According to STAC, the STAC-M3 tests "solutions that enable high-speed analytics on time series data, such as tick-by-tick market data (aka 'tick database' stacks)."

Prior to October 2014, vendor STAC-M3s published by STAC used C/C++ and q exclusively to store and query data in the tick database. But that changed in October 2014, with publication of the first-ever STAC-M3 results featuring a SQL database system. In that implementation, McObject's *eXtreme*DB Financial Edition SQL DBMS deployed on an IBM Power System S824 server with IBM FlashSystem 840 storage set new speed records in 9 of the STAC-M3's 17 response-time benchmarks; delivered record-setting low standard deviation (low jitter) in eight of the 17 tests; and turned in a sum of mean benchmark times (which equates to the best time across all 17 benchmark tests) 1.6x faster than the best previously published STAC-M3 results: 66 seconds vs. 106 seconds.

Should this be attributed to IBM POWER8 "big iron" triumphing over previous Intel-based stacks? STAC followed up less than a month later by publishing another STAC-M3 report in which *eXtreme*DB Financial Edition SQL DBMS on an Intel-based 4-node cluster in the cloud (Lucera) exceeded the previous best non-*eXtreme*DB results for 5 of the 17 tests. A new STAC-M3 report, issued Oct. 29, 2015, shows McObject's SQL DBMS setting new records in six of the 17 tests and cutting time to complete the entire benchmark by an additional 13 seconds (almost 20%) from the October 2014 results.

The fact that a SQL DBMS now "owns" nine of the 17 STAC-M3 tests demonstrates that SQL, in and of itself, is not a performance bottleneck. And this stands to reason. SQL (like q) is merely a language; it is the implementation of the language, and characteristics of the underlying DBMS, that determine performance. So what SQL DBMS characteristics contributed to the record-setting STAC-M3 results?

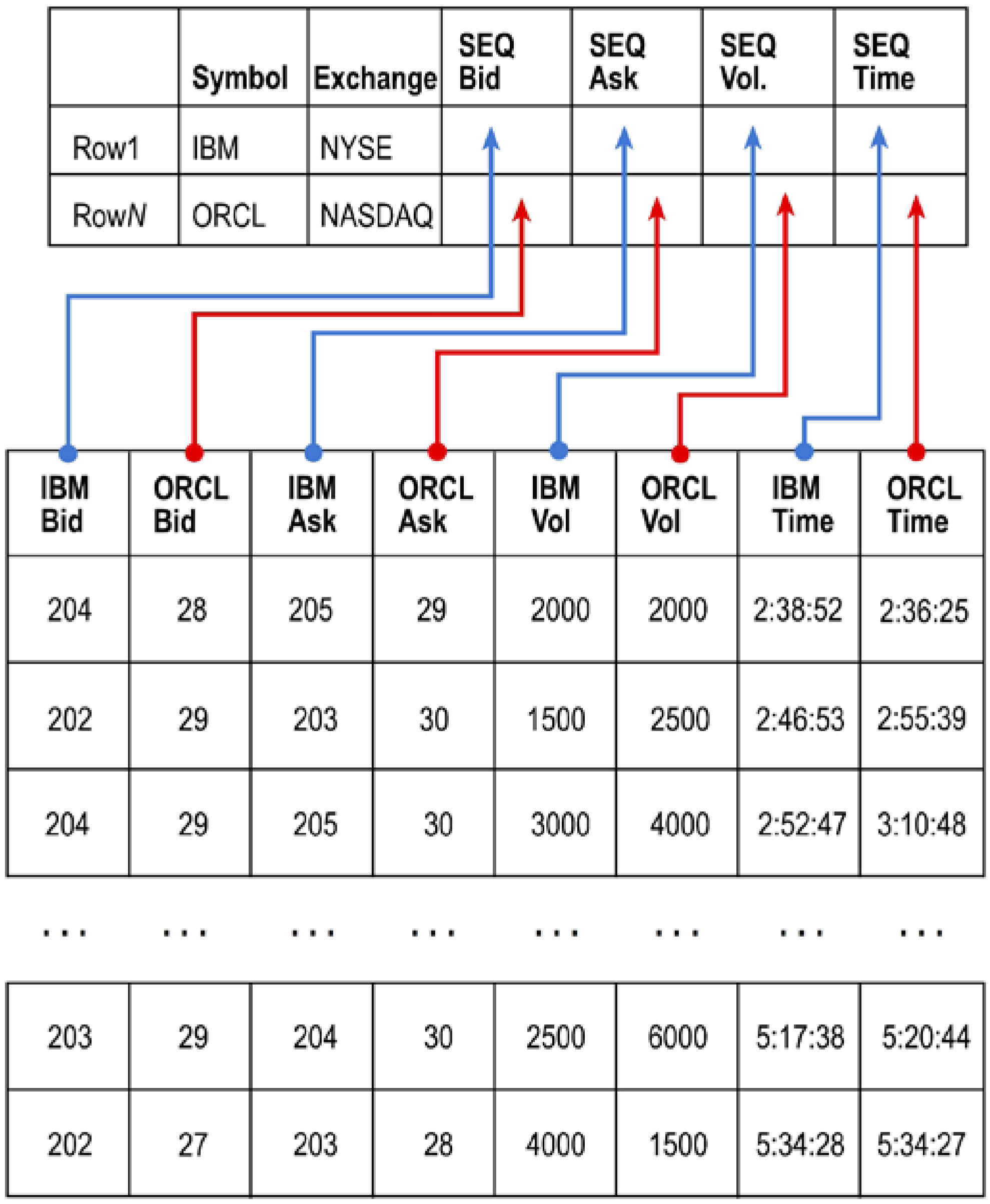
- Columnar layout
- Pipelining
- Distributed Query Processing

Columnar Layout

The vast majority of DBMSs organize data internally in a row-wise fashion. Each table in the database can be visualized like a spreadsheet: many rows, each consisting of many columns. Each row is a record, and each column is an element of that record type. Further, database systems typically store and retrieve pages of data. Page size varies, but a page will have as many rows on the page as the page size will allow – e.g., if a row is 200 bytes wide (irrespective of the number of columns) and the page size is 4096, each page will hold 20 rows.

With these assumptions, if I want to calculate the 5-minute moving average of intraday asking price for a particular ticker symbol, and there are 5,000 ticks per day, the DBMS has to read 250 pages, and transfer 1,024,000 bytes from storage to memory, and then from memory to CPU, to support the calculation. As a data element within a record that contains additional elements (date, time, etc.), the "ask" data will occupy a single column within the database table. If we also assume (reasonably) that the "ask" column is defined as a 4-byte floating point number (i.e., the individual "ask" prices are 4 bytes in size), then the amount of actual data needed for the calculation is just 20,000 bytes (5,000 ticks X 4-bytes). But since the DBMS moves data around in pages of rows (rather than just the column of interest), 1,024,000 bytes of data must be transferred into CPU cache in order to process that 20,000 bytes – 98% of the I/O is wasted!

*eXtreme*DB Financial Edition offers an alternative to this row-based approach. Database designs that use its columnar data layout cause time series data to be stored in columns we call sequences that hang off of their associated rows. For example, as shown in Figure 1, below, the time series data for IBM and ORCL hang off of their respective rows. With this organization, each column (sequence) of the time series is stored on its own page(s). In other words, each database page only contains values of one sequence. Within the same 4,096 byte page size, this columnar data layout can fit 1024 "ask" values, and the DBMS only needs to read 5 pages, or just 2% of the I/O required by the row-wise organization (5,000 ticks / 1,024 per page = 5 pages). And, because only the needed column ("ask") is being fetched and not the entire row, none of that bandwidth is wasted.



	Symbol	Exchange	SEQ Bid	SEQ Ask	SEQ Vol.	SEQ Time
Row1	IBM	NYSE				
RowN	ORCL	NASDAQ				

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

Figure 1.
Traditional DBMSs bring rows of data into CPU cache for processing. But financial data – such as ticks, trades and quotes – are better handled by a column-based layout.

Pipelining

Columnar storage is all well and good if we only have to process a single data element. But financial analysis commonly involves more complex computation. Consider a simple multi-element calculation like:

$$d = a \, X \, b + e$$

When using either traditional SQL, a low-level database interface, or a vector-based language (e.g., q or R), if a, b and c are sequences (vectors) of 100 million values, this operation necessitates creating a temporary result for "a X b" containing 100 million values, then "feeding" this temporary result to the next step of the formula, "+ e," to get the final result of 100 million elements for "d." Obviously, a 100 million value sequence of 4-byte floating point numbers is 400,000,000 bytes – far too large to stay in the CPU cache – so the data generated during this calculation (the interim results) will be shuffled to temporary storage (SQL temporary tables, if it is a SQL DBMS) in RAM.

Or not. A technique we've implemented in *eXtreme*DB called pipelining avoids transferring any of the interim results between CPU cache and main memory. Pipelining brings a page of "a" and a page of "b" into cache and produces one page of the temporary result from "a X b." Then it brings a page of "e" into cache to perform the "+ e" operation and produce a page of the final result "d." At this point, a page of the final result is ready for the application to use (store it back in the database, plot it in a GUI, etc.). When the application iterates off of the last value of the page of "d," the process repeats: fetch a page of "a" and "b," produce a temporary page, fetch a page of "e" and produce a page of "d" and hand off to the application for further processing. This continues until the a, b and e time series are exhausted.

Because we only produce a page at a time of the interim and final result, there's no need to ever move anything from cache to RAM (a 4,096 byte page is plenty small enough to remain in CPU cache). Pipelining means that the output of one function

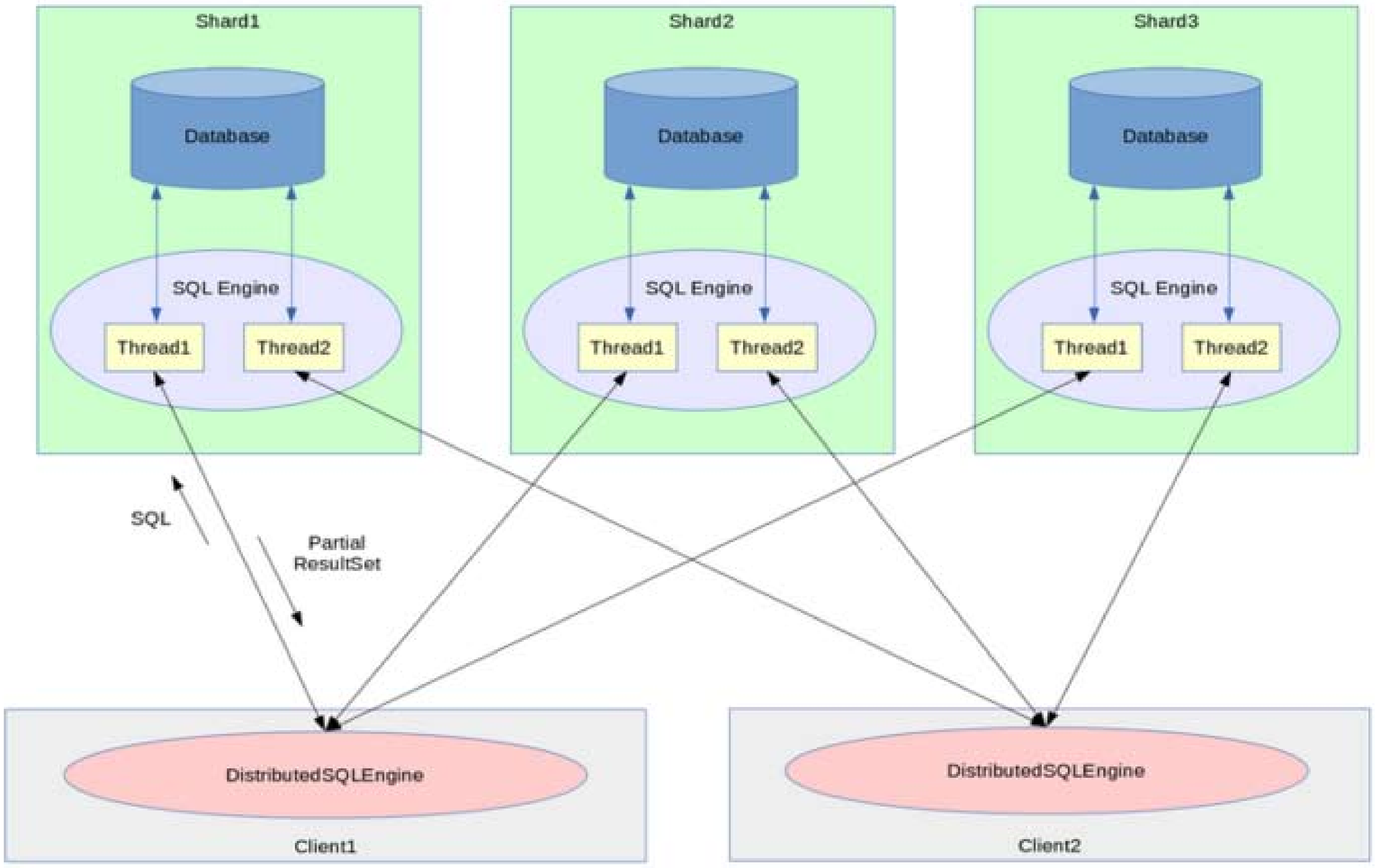


Figure 2.

("a X b") becomes input to the next function ("+ e"). That's very different than having to process the entirety of "a X b" before the next step in the formula. The practical effect is this: The CPU's cache operates on the same silicon and at the same speed as CPU itself, which is at least 2X faster than memory or the bus speed between memory and CPU (often more than 2X). Let's go with 2X, for simplicity. Without pipelining, the interim result has to be written to RAM, and then read back from RAM for the next step. So the performance penalty relative to pipelining is 8X (4 transfers to/from RAM, each of which is 2X slower).

a X b	*1 transfer to write the temporary result to RAM*
+ e	*1 transfer to read the temporary result from RAM*
= d	*1 transfer to write the final result to RAM*
	1 transfer for the application to read/process the final result from RAM

So again, we have 4 transfers, each of which is 2X slower than the CPU. Pipelining eliminates all of these transfers. This is what we call "on-chip analytics" and its speed and performance go well beyond in-memory analytics.

Another benefit is that results are available to the application after processing just the first page of data. In contrast, without pipelining, all of the interim and final results must be processed before the application receives any result.

Distributed Query Processing

The pipelining technique described above enables the DBMS to keep one core of the CPU really, really busy. But how do I leverage the power of 16 cores – or four servers with 16 cores each (64 cores total)?

The answer is sharding and distributed query processing. For optimal performance on one 16-core server, the DBMS partitions the database into 16 shards and "stands up" an *eXtreme*DB server for each shard. The distributed query client submits the query to all 16 servers, each of which produces a partial result set from its portion of the total database. The client assembles the 16 partial result sets into complete results for presentation to the application. In this manner, all 16 cores work equally hard on the problem.

This approach works with any number of servers, and was used in the three record-setting STAC-M3 benchmarks described above. The distributed query client only needs to know each server's IP address and port number, whether the server(s) is local (on the same IP address as the client), or not. Figure 2, above, illustrates.

Importantly, the client application needn't know anything about the shards or servers. It submits exactly the same SQL statement, and receives exactly the same result set, as would be used in a standalone situation. All the scattering of the SQL and gathering and re-assembling of partial results is accomplished by the DBMS's distributed query processing capability.

Tying It All Together With SQL

Does pipelining require pages of arcane code? One striking feature of the technique is that it is accomplished via that most common of data processing commands, the SQL SELECT statement. This is best illustrated with another example.

Two trends commonly followed in technical analysis are the 50-day and 200-day moving averages. When the 50-day moving average crosses below the 200-day moving average, it can be a signal to sell the equity. Conversely, when the 50-day moving average crosses above the 200-day moving average, it can be a signal to buy the equity.

The following example demonstrates how a simple SELECT statement can calculate the 50- and 200-day moving averages of an equity, subtract one from the other, determine when the difference between them crossed over or under zero (meaning, the lines crossed in one direction or the other), and map those crossover points back to the date in the time series at which they occurred. The functions beginning "seq" are from *eXtreme*DB Financial Edition's library of vector-based statistical functions; this naming indicates they are designed to execute over sequences (columns) of time series data:

```
SELECT
  seq_map(closeprice,
    seq_cross(
      seq_sub(
        seq_window_agg_avg(closeprice, 50),
        seq_window_agg avg(closeprice, 200)
      ),
    1)
  ) as Cross
FROM security
WHERE symbol = 'IBM'
```

By nesting the functions in the SELECT statement, we've created a pipeline. The results from the innermost functions (seq_window_agg_avg) will become input to the next outer function in the pipeline (seq_sub) and so on.

Why SQL?

There are some 11+ million programmers earning their living writing code (and many millions more hobbyists). Estimates for the number of "data scientists" range anywhere from 11,400 (seems low) to 1,000,000 (seems high). The point is, there is a large pool of programmer talent that knows SQL. Conversely, only a small number of programmers know any database vendor's proprietary language or API, such as the k or q languages, or *eXtreme*DB's lower-level (non-SQL) API.

In addition to being widely known, SQL is a more productive and readable (and, therefore, maintainable) programming language than a low-level proprietary API or language, just like a higher-level language like Python can be written faster than a low-level language like C. If an organization can get comparable performance using SQL (or performance isn't a consideration), then the advantages of shorter development time and access to a large talent pool carry relatively higher weight.

To put a finer point on it, let's look at two comparisons. Figure 3, below, shows the moving averages example described above in SQL, but implemented in the kdb+ "q" language. Figure 4 illustrates how to calculate the effect of stock splits on historical (end of day) trade data, first in SQL and then in *eXtreme*DB's native API for C/C++.

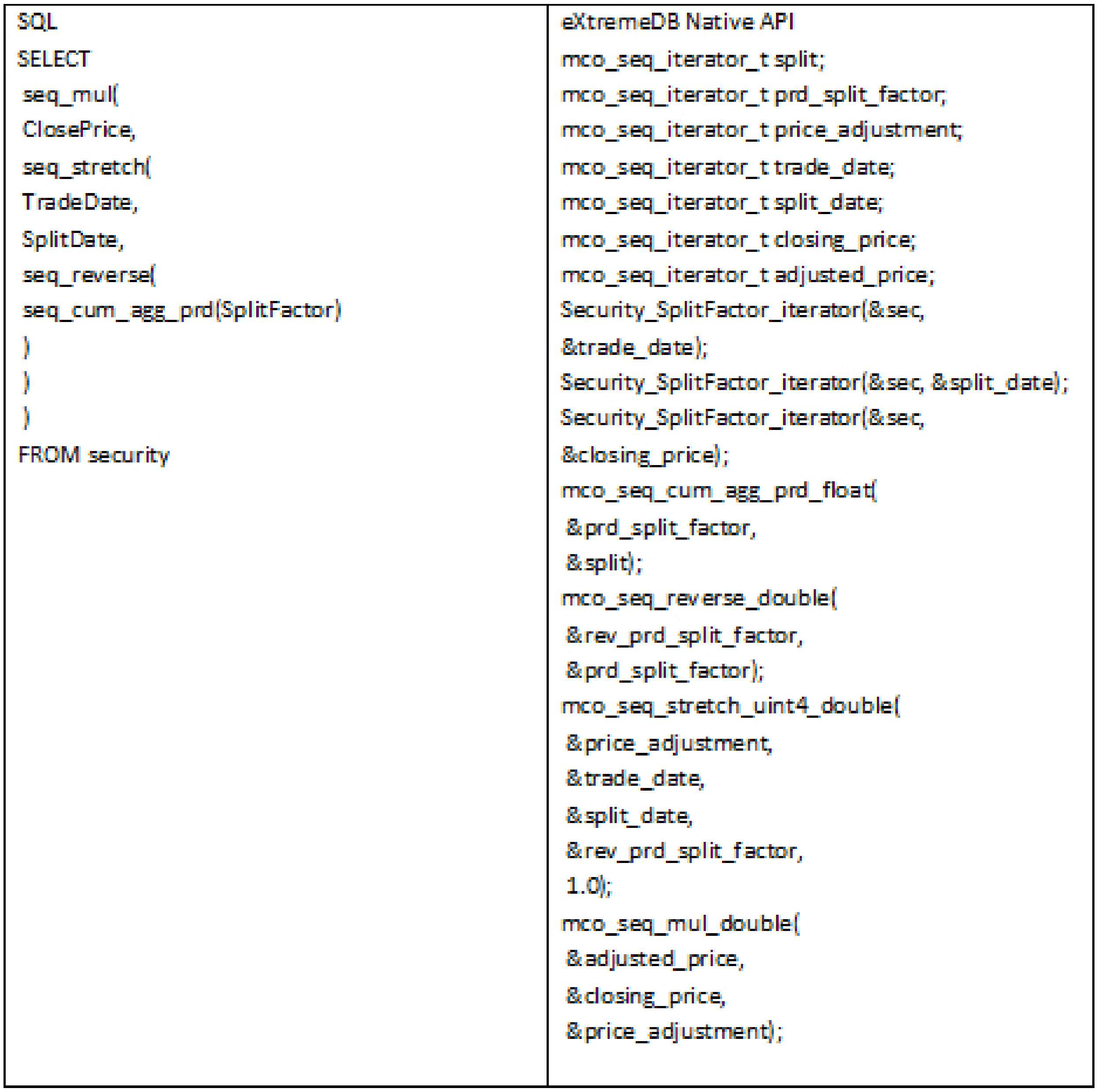
SQL

```
SELECT
 seq_map(closeprice,
 seq_cross(
 seq_sub(
 seq_window_agg_avg(closeprice, 50),
 seq_window_agg_avg(closeprice, 200)
 ),
 1)
 ) as Cross
FROM security
WHERE symbol = ‘IBM’
```

q

```
/ Make Fake Table

ts:enlist 1

do[999;ts: ts, (last ts) +1]

dates: 2004.01.01 + ts

prices: 100 + (1000?3.0)-1.50

pxtable:([date:dates];px:prices)


/ Create mvavg's

MVAVG50:50 mavg pxtable

MVAVG200:200 mavg pxtable


/ Create Cross Table

MVCRSS:MVAVG200-MVAVG50

/ Create duplicate cross table skewed previously

/ by a day

MVPRVCRSS:prev MVCRSS


/ rename prevcross's columns

MVPRVCRSS:`date`prevpx xcol MVPRVCRSS


/ merge cross with previous cross, you should
only

/ see one date column on the left as they are

/ keyed. possibly a visual bug but it did merge

/ correctly
```

Figure 3.



SQL	eXtremeDB Native API

SELECT
 seq_mul(
 ClosePrice,
 seq_stretch(
 TradeDate,
 SplitDate,
 seq_reverse(
 seq_cum_agg_prd(SplitFactor)
)
)
)
FROM security
	mco_seq_iterator_t split;
	mco_seq_iterator_t prd_split_factor;
	mco_seq_iterator_t price_adjustment;
	mco_seq_iterator_t trade_date;
	mco_seq_iterator_t split_date;
	mco_seq_iterator_t closing_price;
	mco_seq_iterator_t adjusted_price;
	Security_SplitFactor_iterator(&sec,
	&trade_date);
	Security_SplitFactor_iterator(&sec, &split_date);
	Security_SplitFactor_iterator(&sec,
	&closing_price);
	mco_seq_cum_agg_prd_float(
	 &prd_split_factor,
	 &split);
	mco_seq_reverse_double(
	 &rev_prd_split_factor,
	 &prd_split_factor);
	mco_seq_stretch_uint4_double(
	 &price_adjustment,
	 &trade_date,
	 &split_date,
	 &rev_prd_split_factor,
	 1.0);
	mco_seq_mul_double(
	 &adjusted_price,
	 &closing_price,
	 &price_adjustment);

Figure 4.

Conclusion

Capital markets technology must digest, sort and analyze more data, and do it faster, than ever before. To meet this need, DBMSs for real-time financial systems have cast a wide net in seeking performance gains. Some improvements are internal to the DBMS, such columnar handling of data. Others optimize the DBMS's interaction with external hardware such as networking and storage. The techniques presented above are a hybrid, leveraging both vector-based processing and columnar data layout (internal improvements), but also streamlining DBMS interaction with its hardware environment, namely the CPU cache.

Multi-core CPUs also present DBMSs with the opportunity to ramp up performance, via horizontal partitioning, as discussed above. Because SQL client/server DBMSs typically include robust built-in network communications mechanisms, they can be especially well-suited as the starting point for solutions that exploit sharding's promise. The resulting high performance, along with SQL's high development productivity and wide familiarity among IT professionals, make a strong case for SQL having a bright future in capital markets systems.