Timeseries Analysis using AQuery to Make Your Fortune

José Cambronero, Dennis Shasha
{ jpc485@nyu.edu, shasha@cims.nyu.edu }

ABSTRACT

The very successful relational model is poorly suited for analysis that requires order. The reason is that the basic data type of the model, the relation, is a set of unordered rows [3].

AQuery embodies a modest syntactic and semantic extension to SQL-92, which shows that the unordered aspect of the relational model is not critical for simplicity and in fact needlessly limits the expressive power. AQuery provides the linguistic constructs to express queries on time series and other ordered data easily and intuitively.

Our demonstration proposal consists of a text/visual interface that will allow demo visitors to explore trading strategies over time series of historical financial data. The reader can find a short video showing how a visitor will interact with our demo here.

1. INTRODUCTION

Order occurs naturally in many fields and by extension in their data. For example, stock prices in finance are observed as sequences, where the order is determined by natural time. In networks, the order could be based on the time order of messages between pairs of communication switches. The demo will focus on finance, so do our examples.

Consider the table $prices(Date, EndOfDayPrice)$. Calculating, say, the three-day moving maximum end-of-day price is a linear time operation if the underlying data is sorted by $Date$. Similarly, establishing the day-over-day change in $EndOfDayPrice$ benefits from a date-ordered table.

Furthermore, it is often the case that these queries must be performed group-by-group. For example, if we extend our table $prices$ with stock ticker information to become $prices(ID, Date, EndOfDayPrice)$ we might want to calculate the moving maximum price for each distinct stock ID.

The complexity of queries increases when we want to consider different windows of ordered data. Consider a common finance query: compute moving averages of different window sizes and identify points in which they cross. A common technical trading strategy suggests buying shares for stocks when a shorter-window moving average price crosses above a longer-window moving average price and selling when the opposite occurs.

Common finance queries include the technical trading strategy just described to pairs-trading based on correlations to a moving-window calculation of variance of returns (to capture the time-varying properties of volatility [14]).

2. EXPRESSING ORDER INTUITIVELY

Traditional relational database systems provide little support for order, with extensions such as ORDER BY being little more than an afterthought. Commonly, developers access their data in a relational system and use a separate language to process it (commonly, R, Python, Matlab, or even Java).

We assert (we are not alone in this: [13] [9] [6] among many other excellent works) that rethinking the relational model to allow order to be treated as a first class concept could open these applications to relational-like systems without unduly complicating the query language.

AQuery, described in [7], provides a natural integration of order in its design and language and thus allows users to express complicated order-related queries with little change to the SQL syntax they are accustomed to.

For example, consider querying $prices(ID, Date, EndOfDayPrice)$ to calculate the 12-day moving variance in returns for various stock tickers. This operation can be critical for financial modeling, as time-dependent variance effects must be accounted for appropriately [4].

The SQL-99 formulation (extended with a built-in VARIANCE function) is shown in Figure 1.

By contrast, the AQuery formulation of that same moving variance query is shown in Figure 2.

AQuery adds an **ASSUMING** clause to the query that says, intuitively, "order this table by Date". The query then uses a built-in function `ratios`, which uses this order to calculate returns along with a variance calculation, `vars`, of moving-window size 12.

The resulting query is, we believe, easy to read for any proficient SQL-92 programmer, whereas the SQL-99 query is challenging to read. Fortunately, this ease of expression comes with a *gain* in performance.

3. AQUERY BACKGROUND

SIGMOD '16 San Francisco, CaliforniaUSA

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

```
SELECT ID, Date,
  VARIANCE(rets) OVER (
        ORDER BY Date ROWS BETWEEN 11
            PRECEDING AND CURRENT ROW
        ) as mv
  FROM
  (SELECT
    curr.Date, curr.ID,
    curr.EndOfDayPrice /
    prev.EndOfDayPrice - 1 as rets
    FROM
    prices curr LEFT JOIN prices prev
    ON curr.ID = prev.ID
    AND curr.Date = prev.Date + 1)
    GROUP BY ID
```

Figure 1: Moving variance query in SQL-99

```
WITH
  variances(Date, ID, mv) AS (
    SELECT Date, ID,
    vars(12, ratios(1, EndOfDayPrice) - 1)
    FROM prices
    ASSUMING ASC Date
    GROUP BY ID
  )

SELECT * FROM FLATTEN(variances)
```

Figure 2: Moving variance query in AQuery

Arrables ("array-tables") consisting of a set of labeled ordered columns (AQuery is fully vertically partitioned) are the primary data structure in AQuery. Because each column is an array, the system can easily index into them at arbitrary positions. In order to work with ordered data all a programmer needs to do is state the expected order using the **ASSUMING** clause, as done in the previous moving variance example, and then choose an appropriate built-in (or user-defined) aggregate.

Figure 3 shows a graph for the query in Figure 2, run on a subset of stock tickers, and downloaded from our visualization interface (described in Section 5.1).

The returns calculation is common in finance, so the `ratios` function divides the value at position i in the array by the value at position $i - x$, where x is the first parameter to `ratios`, with the value at position 0 assigned NULL by definition. Thus, a stock that rose by 10% since yesterday would have a return of 1.1 for that day pair.

The `vars` function operates over a moving window whose size is provided by its first parameter and calculates the variance within each window.

Because moving aggregate queries are so common, AQuery offers `moving(f, w, a)` that applies any function (including user-defined functions) f over moving windows of size w of an array a.

The AQuery system makes use of two performance optimizations to account for order: (i) order only the columns needed by a query and (ii) order them only after selections (with some exceptions due to pre-existing indexes).

3.1 Statistical Arbitrage: pairs trading

We now turn to a different query: correlation pairs. A common trading strategy can involve identifying pairs of stocks (or other instruments) whose returns have historically

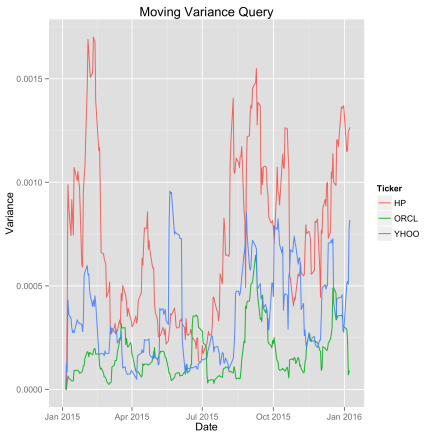

Figure 3: Moving variance calculation for technology stocks (window size 12)

```
WITH
  stocksGrouped(ID, Ret) AS (
    SELECT ID,
    ratios(1, EndOfDayPrice) - 1
    FROM prices
    ASSUMING ASC ID, ASC Date
    WHERE Date >= max(Date) - 31 * 6
    GROUP BY ID)

  pairsGrouped(ID1, ID2, R1, R2) AS (
    SELECT st1.ID, st2.ID,
    st1.Ret, st2.Ret
    FROM
    stocksGrouped st1, stocksGrouped st2)

SELECT ID1, ID2,
cor(R1, R2) as coef
FROM FLATTEN(pairsGrouped)
WHERE ID1 != ID2
GROUP BY ID1, ID2
```

Figure 4: Calculating correlation pairs in AQuery

been highly correlated but have recently become decorrelated. For example, suppose that A and B have had closely correlated returns for years, but recently A's returns increase a lot while B's returns increase much less. The strategy is then to buy B and sell A assuming that their returns will return to correlation. This strategy is commonly known as pairs trading [15]. For our example, we consider calculating the correlation in returns between 10 different stocks over the last 6 months in Figure 4.

We can take advantage of the fact that arrables can be grouped along one or more dimensions and manipulated accordingly as nested arrables. This allows us to cross the two grouped arrables to generate all pairs (including redundant ones) prior to calculating the correlation for each. We also declared the order expected for our analysis: ascending by identifier and by date within that. Our visualization interface gives the results in Figure 5.

3.2 Technical Trading Query

Recall that a simple trading strategy consists of buying the stock when the 5-day average crosses above the 21-day average and selling when the 5-day average crosses below the 21-day average. Figure 6 displays its AQuery formula-

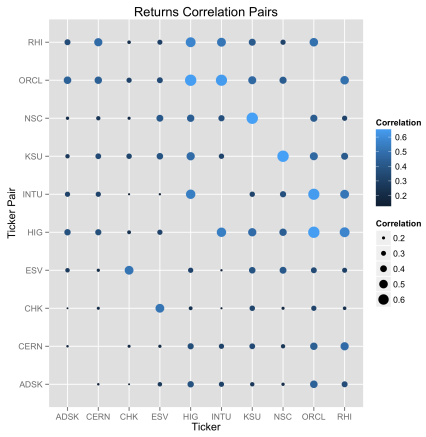

Figure 5: Correlations over pairs of closing returns over the last 6 months

```
WITH
  mavgPrices(Id, TradeDate, ClosePrice,
      m21, m5) AS (
    SELECT
    Date
    avgs(21, EndOfDayPrice),
    avgs(5, EndOfDayPrice)
    FROM prices
    WHERE ID=''HP''
    AND Date >= ''2014-01-01''
    ASSUMING ASC Date)

SELECT
Date, m21, m5,
(prev(m5) <= prev(m21) & m5 > m21) as Buy,
(prev(m5) >= prev(m21) & m5 < m21) as Sell
FROM mavgPrices
```

Figure 6: Crossing moving averages trading in AQuery

tion for a single stock. As in prior queries, this query uses the ASSUMING clause in combination with built-ins such as `prev`, which semantically "shifts" an array back by 1 position.

4. AQUERY: PERFORMANCE

AQuery provides not only a intuitive way to query ordered data but also a performant one. In order to substantiate this claim, we compared AQuery to other excellent established alternatives. Specifically, we compare to Python Panda's library [8], MonetDB with embedded Python [10], and Sybase IQ [12]. We compare these on Sybase IQ's financial benchmark [11].

The benchmark considers a series of queries that reflect common challenges in financial analysis. We performed this analysis repeatedly with randomly generated data. We tested queries with data of 100K, 1M, and 10M observations, along with randomly generated query parameters as appropriate. We focus on the average response time for the various systems across each query.

Experiments against Pandas and MonetDB were run in a single-user setting on a MacBook Air with a 2-Core 1 .7 GHz Intel Core i7 processor, with 8GB of memory. The Sybase IQ comparison was performed on a multi-user linux system with 4 16-Core 2.1 GHz AMD Opteron 6272 processors, with

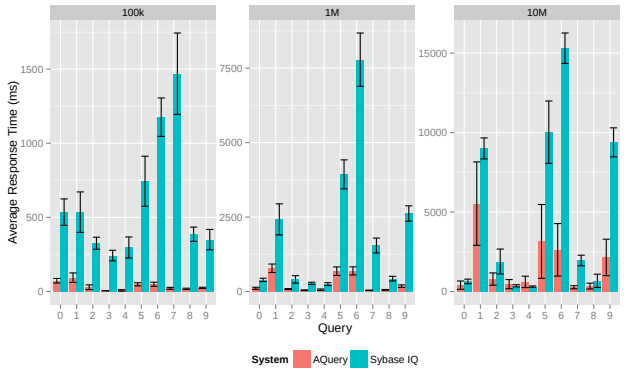

Figure 7: Sybase IQ Comparison on Sybase IQ Financial Benchmark: With 100K and 1M rows, AQuery outperforms Sybase IQ in all of the queries evaluated. At 10M rows, performance is a bit more varied, with larger standard errors, but on average AQuery is faster in 8 of the 10 benchmark queries.

256GB of memory.

Queries include summarizing prices across various time periods for multiple tickers, price/volume adjustment for stock events, moving average calculations (and their intersections), correlation pairs and others. Figure 7 summarizes our results compared to Sybase IQ, showing that in the vast majority of cases AQuery outperforms by orders of magnitude (the scale is logarithmic). We observe similar performance gains when compared to Pandas and MonetDB, but omit those figures here for lack of space.

5. DEMONSTRATION PROPOSAL

5.1 Visualizing Queries and Finding Good Trading Strategies

Our demonstration will provide demo visitors with a simple graphical interface for visualizing the results of various financial queries on real data, such as end-of-day stock price and volume information. Visitors will try to find good trading strategies by writing queries in AQuery.

The predefined set of queries will include the two we have mentioned before:

- Pairs trading: Explore correlations in price returns for various stocks across time and trade when previously correlated pairs go out of correlation.
- Technical trading: Buy the stock when the x-day average crosses above the y-day average and sell when the x-day average crosses below the y-day average.

The visitors will be able to interact with the plotting interface so as to visualize results (such as running profits) from the predefined queries and explore new results with queries and plots of their own concotion.

The interface is a simple browser-based application using R's Shiny library [2] to construct the interface and R's ggplot2 library [17] to provide flexible and elegant plotting.

For example, a visitor could pick the moving average-based technical trading strategy described. The AQuery code associated with the strategy is transparently displayed

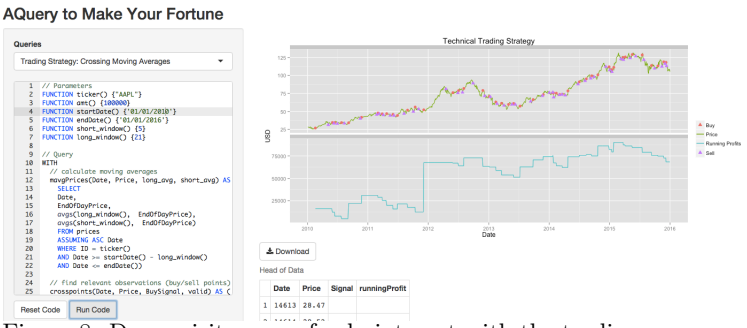

Figure 8: Demo visitors can freely interact with the trading strategies' code and plots: here a technical trading strategy

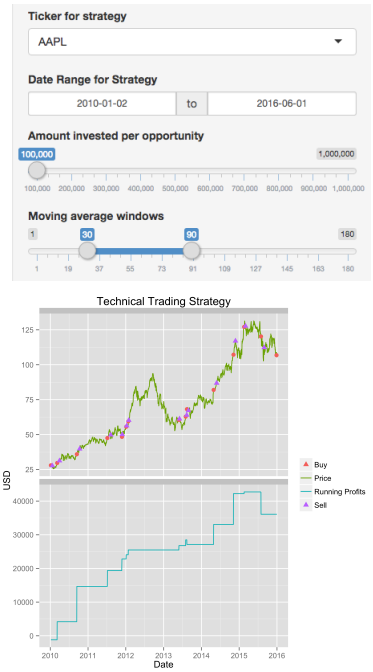

Figure 9: Demo visitors can use the widgets and the code editor. Here the moving windows have been extended and the aggregates have been switched to moving variance

in the interface using the shinyAce [1] editor. During inspection of the code, the visitor can freely interact with it by editing directly or making use of the interface widgets, such as sliders.

Consider a technical strategy with 5-day and 21-day moving averages, looking for trades in Apple stock. Figure 8 shows a screenshot of the interface with 2 plots. The top plot displays the price and buy/sell opportunities. Similarly to other strategies, the bottom plot displays a running profit/loss curve associated with each buy opportunity. So for example, a value of $40,000 indicates that following the strategy has generated that amount of profit up to that trade date (ignoring transaction costs). In the top plot, we can see that there are plenty of buy/sell points, as our visitor has chosen a relatively active trading approach by specifying moving averages with short windows.

However, the visitor could now edit the code or use our predefined parameter widgets to extend this to 30-day and 90-day moving averages. Because there is nothing special about a moving average in AQuery, our visitor might now decide to experiment with moving variance (something a volatility trader might be interested in!). Figure 9 shows the trading results for the new strategy. It is clear from the fewer buy/sell points in history that this is a less active strategy. In real life, this would reduce trading costs.

We will walk visitors through similar exercises and explore both the visualization interface and query code. As mentioned in the abstract, the reader can find a short video showing the interface in action here.

6. REFERENCES

[1] Jeff Allen. shinyace. https://github.com/trestletech/shinyAce, 2013.

[2] Winston Chang, Joe Cheng, JJ Allaire, Yihui Xie, and Jonathan McPherson. *shiny: Web Application Framework for R*, 2015. R package version 0.12.0.

[3] Edgar F Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[4] Frank J Fabozzi, Sergio M Focardi, and Petter N Kolm. *Financial modeling of the equity market: from CAPM to cointegration*, volume 146. John Wiley & Sons, 2006.

[5] Kaippallimalil J Jacob and Dennis Shasha. Fintime - a financial time series benchmark. *SIGMOD Record*, 28(4):42–48, 1999.

[6] M Kersten, Ying Zhang, Milena Ivanova, and Niels Nes. Sciql, a query language for science applications. In *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, pages 1–12. ACM, 2011.

[7] Alberto Lerner and Dennis Shasha. Aquery: Query language for ordered data, optimization techniques, and experiments. In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pages 345–356. VLDB Endowment, 2003.

[8] Wes McKinney. *Python for data analysis: Data wrangling with Pandas, NumPy, and IPython.* " O'Reilly Media, Inc.", 2012.

[9] Wilfred Ng. An extension of the relational data model to incorporate ordered domains. *ACM Transactions on Database Systems (TODS)*, 26(3):344–383, 2001.

[10] Mark Raasveldt. *Embedded Python/NumPy in MonetDB*. MonetDB, 2015 (accessed November 06, 2015).

[11] SAP. *Sybase RAP: User Guide Historical Market Data Queries*, 2008 (accessed November 8, 2015).

[12] SAP. *Introduction to SAP Sybase IQ: SAP Sybase IQ 16.0*, 2013 (accessed November 8, 2015).

[13] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. *SEQ: Design and implementation of a sequence database system*. Citeseer, 1996.

[14] Ruey S Tsay. *Analysis of financial time series*, volume 543. John Wiley & Sons, 2005.

[15] Ganapathy Vidyamurthy. *Pairs Trading: quantitative methods and analysis*, volume 217. John Wiley & Sons, 2004.

[16] Arthur Whitney. *Abridged Q Language Manual*, 2009 (accessed November 6, 2015).

[17] Hadley Wickham. *ggplot2: elegant graphics for data analysis*. Springer New York, 2009.