The Virtues and Challenges of Ad Hoc + Streams Querying in Finance*

Alberto Lerner[†]
Ecole Nationale Superieure
de Telecommunications
Paris - France
lerner@cs.nyu.edu

Dennis Shasha
New York University
New York - USA
shasha@cs.nyu.edu

Abstract

Financial trading strategies are based on queries over time-ordered data. The strategies value very recent data over older data, but require information about older data to avoid making poor decisions. One can imagine a streaming architecture that keeps a synopsis of older information available for many possible queries, but this may be too crude - and too expensive – an approximation of the query requirements. Instead, we show several fundamental query types for which traders would prefer to issue an ad hoc query Q and then allow updates to change the answer to Q over time. To make the ad hoc portion fast, the architecture puts historical data into a well-structured form (organized by security and time) to support rapid querying whereas recent data is ordered by time alone. Periodically, some of the older recent data is moved to the historical data structures. The net effect is to allow queries to look at very recent and less recent data efficiently and only when needed. Several new research issues arise in this setting.

1 Nature of Financial Data

Consider data coming from trading activities of equities (stocks), in particular electronic trading markets such as the NASDAQ. Such markets generate large volumes of data, in bursts that can achieve up to 4,200 messages per second [3]. Those messages present momentary opportunities to make profits if they can be processed in an appropriate way. Before explaining those strategies, let us consider a slightly simplified example of the mechanics of trading:

Bob decides to move some savings into company A's stocks. He calls his broker and places an order to buy 1000 of A's shares at a maximum price of $12.00. This is called a *bid price*. Bob's broker uses the stock market's trading system to broadcast Bob's bid. When it hits the market, the best offer for A's shares, called its *ask price*, is $12.03. Alice wants to sell some of her shares of A. She instructs her broker to sell 1000 of her A's shares at market value. When her offer meets Bob's bid, a trade is done. Each bid, ask, and trade is called a "tick." There

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

*Work supported in part by U.S. NSF grants IIS-9988636 and N2010-0115586.
[†]This work was done while this author was visiting NYU

are up to about 100 million ticks per six-and-a-half-hour day. The time series in this context involve prices and volumes (number of shares) of bids, asks, and trades.

These series can be obtained by a subscriber in distinct levels of detail [6]. Strategies use each level of detail differently. For instance, applications called *Level I* are real-time displays of best (maximum) bids, best (minimum) asks, and the last trade price. *Level II* displays show more in-depth information by presenting the top 5 best levels of bids and asks with their respective market participants. Such information is useful because it can indicate the level and depth of supply and demand.

Those "displays" present real-time summaries of trading activities. Alarms (continuous queries) can be set to detect very simple conditions, such as when a price is the maximum price of the day. We show in the next section that the technical analysis of the finance streams involves more complex queries in addition to these simple alarms.

2 Technical Analysis Strategies

Technical analysis is the activity of making buy/sell decisions based on the time course of a stock, perhaps with respect to other stocks. Here we give a few techniques used by traders [5].

2.1 Intraday

Intraday trading attempts to uncover momentary opportunities (at most a few minutes old) to make small profits. Here are some example opportunities:

"Scalper" trading tries to find a tight interval within which the ask and bid prices are currently oscillating and buys or sells at mid points. For instance, a "scalper" would try to buy A's share if it were quoted at $11.98, before Bob's had the chance to hit the market, if such an ask price were announced. If the "scalper" were fast enough, he could have even sold these shares to Bob. A human scalper can do a few hundreds of these operations in a day.

Pairs-trading tries to take advantage of shares that are usually impacted by the same effects. Simplifying to the essence of the idea, if two stocks tend to differ by no more than $20 and the higher one goes up whereas the other goes down, then sell the first and buy the second. Sell out when the difference returns to normal.

Note that in the above cases, very recent history (i.e., a few seconds to a few minutes old) is vastly more important than older history. In the next example not-so-recent data is involved.

"Hammer" discovery. Market Makers are large holders of shares and are thus capable of buying or selling heavily. Whenever they do so they can exercise some control (their "hammer") over the quotes of a given security. To avoid calling attention to themselves, they partition the volumes they are moving into a number of small trades to avoid paying more when buying or receiving less when selling. Savvy investors try to infer such moves by trying to identify any hammer-type movement – which can be diluted over a period ranging from minutes to hours – and profit from it by bidding against the hammers (e.g., buying when the hammers are buying).

2.2 Long-Term

At the other end of the continuum of trading strategies come those that consider long periods of price series. Here are a few examples:

Crossing averages. Moving averages are capable of smoothing the volatile price curves and exposing underlying optimistic or pessimistic sentiment. For example, whenever a short term trend curve (a moving average over a few days) climbs above the long term one (a few weeks or months) one, technical analysts will suspect that the stock will move up soon.

Breakout in support-resistance curves. Some view a trading market as a continuous fight between buyers and sellers, and attribute to them the power to control the range within which prices oscillate. The basic idea is that buyers will predominate when the price is cheap enough (at or below its "support level") and sellers will predominate when the price rises above its "resistance level." Occasionally, of course, the price exceeds this range, an event that may be of considerable interest to a trader.

2.3 Characterizing Queries in Technical Analysis

Underlying each trading strategy are queries over the time series. Long-term analysis have little need for intraday trading data. A salient aspect about these queries is that fact that they often depend on order (e.g., moving averages, or deltas of prices on time-ordered series). Therefore, long-term queries can be implemented using order-dependent query languages over static data. Our language AQuery [1] is an example of such a language.

In turn, intraday data requires several kinds of queries:

- Continuous: every time an element of data such as a quote or a trade arrives, the query has to be re-evaluated. Typical applications: scalper and pairs trading.
- Periodic queries: Continuous queries over fast-paced streams may overflow an analyst with results that may not be needed as soon as they are processed. Periodic queries allow data to accumulate for a fixed amount of time and then issue the query up to the latest time point. A typical application is a query looking for hammer activity in a given stock. Periodically, perhaps on demand, we want to see if the hammer is active.
- Ad-hoc: just prior to doing a trade, we may want some longer term analysis. For example, we may have heard rumors of a hammer and then verify it.

3 A Detailed Query Example

Suppose a trader believes there might be a Hammer trying to acquire shares of ACME, but is not yet sure. He/she issues an ad hoc query to find out. The query essentially asks whether any market maker has many *inside bids* in the recent past. An inside bid is the highest bid at a given time.

If the query were on purely time ordered data, the ACME data would be mixed in with 10,000 other stocks. This would require a scan of all that data. The query would go much faster, however, if ticks were organized by stock and then by timestamp within each stock. Such data reorganization is the key to answer our query within a reasonable amount of time. However, it may not be feasible to do such a reorganization in real-time. Let us see how a periodic reorganization can be done instead.

Let the ticks' schema be Ticks(sID, MMID, price, volume, type, timestamp) where sID is a security's identifier; MMID, the market maker behind this tick; type is either 'bid', 'ask', or 'trade'; price and volume are associated with this tick; and timestamp is the precise instant the tick enters the trading system. We will be using AQuery, a dialect of SQL that incorporates the notion of order, to show the queries discussed here. We will be explaining its features at they appear. A more complete reference about that language (its underlying data model and its optimization) can be found at [1]. AQuery's formulation of the hammer discovery query looks like:

```
WITH
   MaxPrice(insideBidPrice, timestamp) AS
   (SELECT  maxw( range(90,timestamp) , price ), timestamp
   FROM     Ticks
            ASSUMING ORDER timestamp
   WHERE    timestamp > now() - 30 minutes
            AND type = 'bid'
            AND sID = 'ACME')
SELECT  MMID, count(*)
FROM    Ticks t, MaxPrice mp
WHERE   t.timestamp = mp.timestamp
        AND t.timestamp > now() - 30 minutes
        AND type = 'bid'
        AND sID = 'ACME'
        AND price = insideBidPrice
GROUP   BY MMID
```

Figure 1: The "Hammer Discovery" query.

The first half of the query computes ACME's inside bid prices for the last 30 minutes, considering that a bid has a 90 seconds lifespan[1]. The WITH is a construct borrowed from SQL:1999 that allows a query to define a "local" view [2]. The ASSUMING ORDER clause is AQuery's and is used to enforce a given sort order after the FROM clause is processed. Thus, in the first half of the query, Ticks can be assumed to be in timestamp order. All subsequent clauses in the WITH query can count on and preserve that ordering. AQuery also provides convenient ways of manipulating date and time data, e.g., the function now() that returns the system's current timestamp, or the literal '30 minutes' as a way to express a time value. Finally, AQuery's semantics is column-oriented, meaning that instead of having variables that range over collections, an AQuery variable is bound to an entire collection – an array to be more precise – at once. For instance, the function 'range(90, timestamp)' is called only once and it is passed the entire column timestamp as its second argument. Range takes a value v and an ordered vector c whose element type is the same as v's, and returns a vector $\text{range}_{v,c}[i] = w[i]$ where $w[i]$ is the minimum index such that $c[i] - c[i - w[i]] <= v$. In our case, range() returns for each timestamp t, the number of previous ticks the window should contain if one wanted that window to span from t - 90 seconds up to t [2]. The function 'maxw()' takes a vector w of window ranges and a vector c and computes $\text{maxw}_{w,c}[i] = max(c[i - w[i]]..c[i])$. The main query identifies the Market Makers that matched any inside bid at any given point and counts how many times that happened for each distinct Market Maker.

Now, if one issues the hammer discovery query late in the day, available ticks over the last several hours should be looked at. To make that task efficient, we keep recent and historical ticks in different structures. We describe those structures as (maybe materialized) views over the stream. Recent data is maintained in a "RECENT" portion of Ticks, which is defined as a view over Ticks, as seen in Figure 2.

Data arriving at the system is immediately available at the RECENT view. But the latter necessarily have an "aging" predicate that determines that data must be purged when the aging predicate is false. In TicksRecent, this predicate is 'timestamp => now() - 5 minutes.' Aged RECENT data is moved to HISTORICAL views. See Figure 3. Moving the data efficiently is an implementation issue in its own right – when to do it, how to maintain data while it's being done and so on. We return to this issue later.

Note that RECENT and HISTORICAL views may organize data differently. In this particular case, they differ in the way they sort it: whereas recent Ticks are sorted by timestamp, older ones are organized by security ID and sorted by timestamp for each such ID. Intuitively, to answer the hammer search query, the system would

[1] A bid's lifespan can vary depending on several factors, including which system was used to place it and even which time of the day it was placed [4], so this is a slight simplification.

[2] Windows in AQuery are mere aggregate function arguments. Some other languages handle them as special language constructs, confined to specific parts of a query (e.g., valid only in the SELECT clause). We argue in [1] for the benefits of the former over the latter.

```
CREATE RECENT VIEW TicksRecent AS
SELECT sID, MMID, price, volume, type,
       timestamp
FROM   Ticks
       ASSUMING ORDER timestamp
WHERE  timestamp => now() - 5 minutes
```

Figure 2: Ticks' RECENT View

```
CREATE HISTORICAL VIEW TicksHistorical AS
SELECT sID, MMID, price, volume, type,
       timestamp
FROM   Ticks
       ASSUMING ORDER sID, timestamp
WHERE  date(timestamp) = today()
       AND timestamp < now() - 5 minutes
```

Figure 3: Ticks' HISTORICAL View

break that query into two parts. While the recent part of it would have to select out 'ACME' ticks among all others, the historical part would be able to select a contiguous extent of 'ACME' ticks. Because data in the historical part is more voluminous, it is important that such a query not require a scan of it all. Figure 4 shows possible query plans for the situation above.

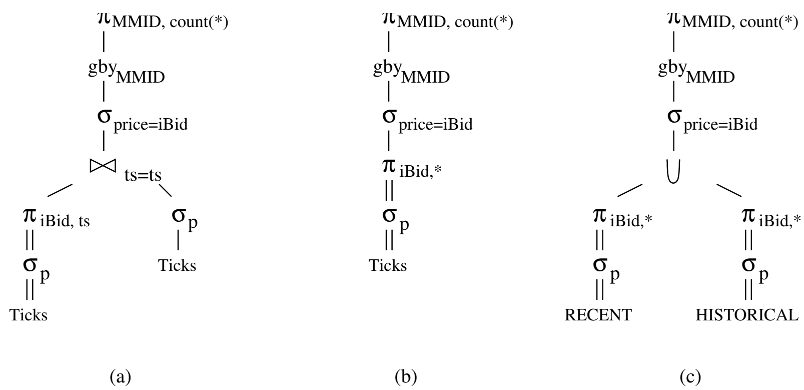

Figure 4: Plans for the Hammer Discovery Query: (a) Initial Plan (b) Optimized Plan over Stream Data (c) Optimized Plan over RECENT + HISTORICAL Data

The plan in Figure 4(a) follows the query's syntax. We abbreviate `date(Ticks.timestamp)=today() AND type='bid' AND sID='ACME'` for p, and `maxw( range( 90, timestamp ), price)`, the inside bid price, for $iBid$. Note that some operators in the plan are connected by double arcs. That signifies that existing order is being maintained by operators. For instance, in the left-side branch of the join, the existing order of Ticks (over timestamp) is preserved. That is necessary, because the calculation of iBid requires data to be in timestamp order. After that computation is done the ordering requirement is drop.

The plan can be simplified. The left-hand side of the join is simply computing a new column, iBid. The join can thus be exchanged for a projection that adds that computed column to the ordered table. The result is depicted in Figure 4(b).

The performance of that plan grows slower as the stream it is acting upon becomes more dense over time (i.e., more ticks per second). The top curve on the graph presented in Figure 5 shows that progression. We are simulating a one-hour-long stream with varying number of ticks per second. The query is required to look at the last 30 minutes ticks only. But, as said before, ACME ticks are mixed within thousands of others. The cost is dominated by the evaluation of predicate p. The reorganization of aged data clearly pays off as the second curve on the graph suggests. But before commenting on those results, let us show how a more elaborate plan can take advantage of the RECENT and HISTORICAL data organizations.

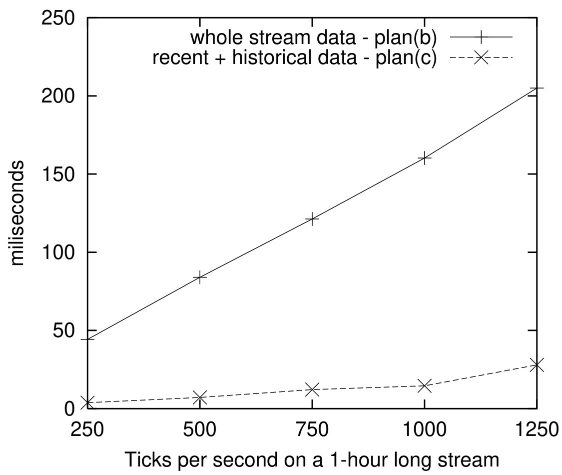

Figure 5: Performance Comparison between Figure 4's plans

The union of RECENT and HISTORICAL ticks contains all the ticks in the original stream. Thus, one can decompose the original query to use the (materialized) views described before. In our present example, it suffices to compute iBid over both the views, take the union of the results and then select which Market Makers matched the inside bids and count how many times per Market Maker. Such a plan is depicted in Figure 4(c).

Such an elaborate plan yields much better performance. Let's see why. The predicate p should be evaluated over both RECENT and HISTORICAL data. RECENT has the same organization as the original stream and so it is expensive to evaluate p over it. But recall that RECENT doesn't keep all the data. Here the aging predicate is discarding data that is 5 minutes old. As a result, there are far fewer rows to look at. Now, those rows go to HISTORICAL when they age but are ordered there by sID and timestamp. Therefore, the conjuncts of p that find ACME's ticks and the desired timespan of ticks can be computed with faster binary searches. That alleviates most of cost of computing p, as those conjuncts are quite selective. The remaining part of the query concatenates the results, computes inside bids and so on, as before. The result: not only does the plan (c) in Figure 4 have a much lower cost than plan (b) – one order of magnitude – but also its time increases slower as a function of stream density.

Of course, this organization has a price, but if the query density is high enough (or the importance of the queries is high enough), it is worthwhile.

This data reorganization rationale can be generalized as we discuss next.

4 A Generalized Streaming Architecture

We now describe an architecture that allows the ad hoc queries to be processed using the two data organizations based on the modules of Figure 6.

Rows arrive as a set of streams. The *dynamic router* taps into those streams and extracts rows' elements that are to be inserted into RECENT views. As its name suggests, the dynamic router is capable of directing a row to its respective RECENT destination. This module and all the remaining ones depicted in Figure 6 have access to metadata (catalog) information.

Eventually rows at RECENT views become aged. Whenever that happens – or at strategic moments (e.g., low system workload) – the *transfer agent* is responsible for purging them from RECENT and inserting them into HISTORICAL views. To avoid concurrent contention, some data may be redundant in the two tables for a time. In that ways, queries will always find all the necessary data.

Ad hoc queries sent to the system are handled by the *ad hoc query processor*. This module is responsible

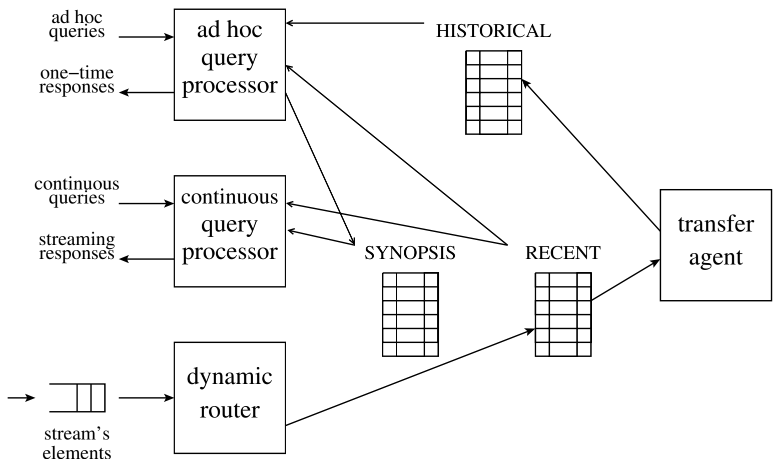

Figure 6: A Generalized Architecture for Ad-Hoc and Continuous Queries

for breaking down an ad hoc query, which is expressed against an integration schema, into component queries that can be executed against RECENT and HISTORICAL data. It then creates a synopsis data structure and passes the query to the *continuous query processor* so the answer to the query can be maintained as updates arrive. When the query ceases to be of interest, then the continuous query processor must be informed and stop processing updates.

5 Summary and Open Problems

Streaming is often viewed as a technology in which one has only one chance to look at data, and limited memory. Thus viewed, streaming is a useful technology for financial databases. Of the kinds of technical queries we've identified, scalping, and breakout are essentially stream queries.

On the other hand, some queries must be done retrospectively after a short term pattern (e.g. of a potential hammer) suggests looking at historical data. Because the data required by such queries is ideally ordered by some other attribute as well as by time, some preprocessing is desirable on most if not all the data. For this reason, we propose a generalized streaming architecture that supports efficient ad hoc queries, but allows classical streaming on recent data. Its main characteristics are:

1. splitting recent data from better-indexed historical data, where the indexing includes data ordering;
2. streaming techniques for the recent data including running synopses to support the maintenance of query responses over time;
3. order-aware query optimization for both recent and historical data.

Such an architecture presents two major challenges that we are actively studying:

- Maintaining the historical data in this useful organization without interrupting concurrent queries requires the avoidance of locks and perhaps the postponement of updates.
- Handing off the processing of continuous queries from the ad hoc query processor to the continuous query processor.

These challenges are real, but far from insurmountable. We foresee the deployment of high performance, order-aware, generalized streaming systems to a trading desk near you in the not-too-distant future.

References

[1] Lerner, A., and Shasha, D., "AQuery: Query Language for Ordered Data, Optimization Techniques, and Experiments." TR2003-836, Courant Institute of Mathematical Sciences, New York University, March 2003.

[2] Melton, J., and Simon, A.R. "SQL:1999 – Understanding Relational Language Components." Morgan Kaufmann, May, 2001.

[3] NASDAQ. "Peak Message Rates fro NASDAQ SuperMontage Data Feeds in December 2002." Nasdaq Vendor Alert #2003-4, January 17, 2003.

[4] NASDAQ. "SuperSOES Frequently Asked Questions." http://www.nasdaqtrader.com

[5] NASDAQ. "ViewSuite Data and NASDAQ SuperMontage: PowerView." http://viewsuite.nasdaqtrader.com/resources/Level2_pres.ppt.

[6] NASDAQ. "Nasdaq Data Feed Products." http://www.nasdaqtrader.com/trader/mds/nasdaqfeeds/feeds.stm