

---

# Processing Real Finance Streams: challenges and techniques

Alberto Lerner<sup>1</sup> and Dennis Shasha<sup>2</sup>

<sup>1</sup> IBM T.J. Watson Research Lab [alerner@us.ibm.com](mailto:alerner@us.ibm.com)

<sup>2</sup> Courant Institute of Mathematical Sciences – New York University  
[shasha@cs.nyu.edu](mailto:shasha@cs.nyu.edu)

## 1 From Packets to Tuples

A Data Stream Management System (DSMS) is essentially a database system that allows one to query data which is not statically stored. One can think of queries as being registered in the system and being run whenever relevant data is seen. For example, suppose that a trader is interested in knowing when any two consecutive IBM stock trades prices differ by more than a given threshold. The trader would register a query that would look at IBM quotes, which would then be used to compute price deltas, which would finally be tested for the threshold. The trader would thus be notified every time an IBM trade passed the test.

The implicit assumption here is that this query has access to a stock's trade stream. This stream is supposed to carry trades in a format convenient for the query.

Real streams do not come in such a convenient format. Stream are structured to transport data efficiently rather than for query convenience. Finance streams such as the NASDAQ feeds are an example [1]. This chapter considers the issue of converting such efficiency-encoded streams into well-behaved and convenient ones. There are several issues all of which present challenges and optimization opportunities.

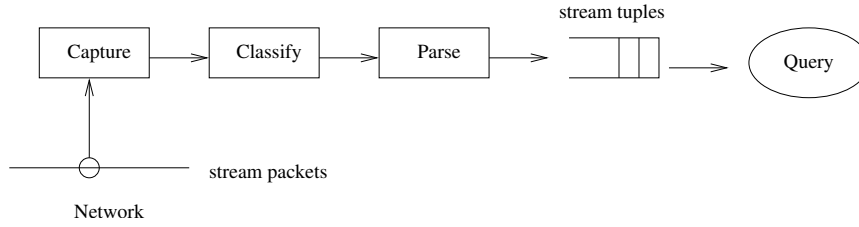
First, the stream data is bursty and voluminous, so getting data into memory is a challenge. Traditional sockets are usually too slow. It may be better to look at every flowing packet regardless of its destination.

The second problem is to demultiplex the feed into the convenient streams of interest. Packet classification must be fast and should require only constant storage.

Finally, feed packets may have a more complex structure than a query would expect. Consider, for instance, the main data feed in NASDAQ that reports on trades, the "UTP trade data feed," or UTDF [1]. Each UTDF packet consists of a block of records, each of which can be of one of 25 possible types. There are six trade-related types alone. One needs to parse such a

packet into tuples – possibly belonging to several streams – before a query can act on the contents.

The architecture 1 we present bridges this gap between real-world streams and the streams that queries are able to process.



**Fig. 1.** An architecture for handling raw streams of data

Data in the architecture flows in a network-to-query direction. The packet-capturing module makes packet contents available to the system. The classifier module determines for each packet the stream (or streams) it is carrying data for. Finally the parser extracts the data from the packet contents and formats it in a query-convenient form.

Optimization information, by contrast, flows in a query-to-network direction. For example, if no query requires a given stream, then the classifier may drop that data as soon as it sees it. Similarly, the optimizer can tell the packet-capturing module to avoid packets addressed to certain ports.

Before presenting the issues facing each component of the architecture, we describe the NASDAQ feeds in some detail for purposes of the concreteness of our running example.

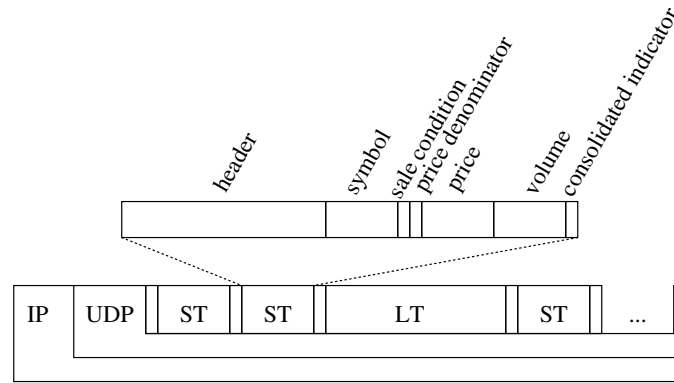
## 2 One Afternoon at NASDAQ

NASDAQ disseminates real-time information through a dozen or so *data feeds* [1]. Feeds may differ by the type of information they carry (mainly trades or quotes) by whether the data is consolidated or not (several quotes levels or just the best ones) or by the markets they cover (NASDAQ or combined).

### 2.1 An Example Feed

The UTDF carries information about NASDAQ-listed securities' trades. Its packets may carry records of 25 possible record types. Records in a packet are separated by special characters. In a given packet there can be not only different types of records but also information concerning distinct securities. UTDF packets are relatively large, usually between 200 and 1000 bytes. Figure 2 depicts a typical UTDF packet. Every UTDF record carries a header that

contains control information such as the record type, sequence number, date and time, etc.



**Fig. 2.** Layout of the main record type on the UTDF feed.

The most common trading record is the “short trade” (ST), but there are also the less frequent records such as the “long trade” (LT), trade correction records, and others. The short trade record has the following fields: symbol of the security that was traded, the price at which the trade took place, and the volume of the transaction. There are three additional flags (1 byte each) in the record: the consolidated indicator field set to 1 whenever the record carries information about several trades, the sale condition is a modifier that denotes the kind of the trade being reported (e.g., regular, split, etc), and the price denominator determines where to place the decimal point in the price field.

## 2.2 Network Profile of NASDAQ feeds

The NASDAQ produces data at highly variable rates. This reflects the frequencies at which quotes and trades hit the market. To do a good job processing the feeds, one must understand their traffic profile. We thus decided to follow a typical afternoon at NASDAQ.

A generous collaborator at Wall Street allowed us access to a router where all the NASDAQ feeds arrive. We have thus recorded the trading session of December 9 2003, between 12:15pm and 2:25pm. That was a fairly undisturbed session.<sup>3</sup>

<sup>3</sup>Alan Greenspan, the chair of the Federal Bank, has been known to cause some disturbance on the trading floor by announcing an interest rate change. Other disturbances may come from news such as unemployment data, quarterly profits reports, political unrest, etc.

The main statistics about the data are presented in table 1. The traffic is composed not only of the UTDF feed described previously but also by at least seven other different streams. These are UQDF, Prime, NQDS, ADAP, Montage, iMQuotes, and TDDS. Please see [1] for a detailed description of those feeds.

**Table 1.** Summary of Traffic Statistics for the Combined Feeds

number of packets	2,942,497
number of bytes	2,100,403,363
time of the session in seconds	7,762.55
average number of bytes per second	270,581.3
average megabits per second	2.06
average packets per second	379
average of bytes per packet	713.81

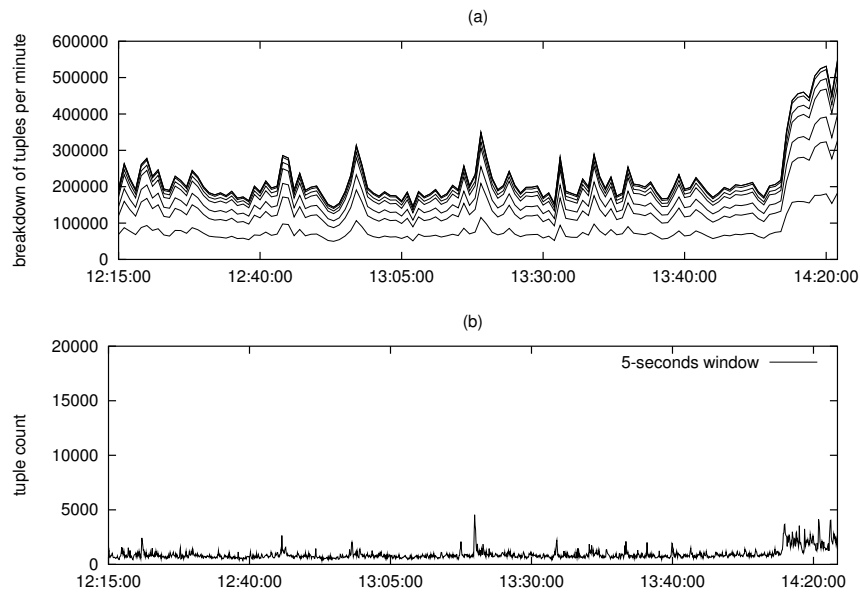
The statistics can be misleading. For one thing, the average number of packets per second and megabits per second may seem low. However, the packets carry several tuples at once. It is actually quite common to find packets carrying more than 20 tuples at once. A system processing the streams has to cope not only with the packet arrival rate but also with the actual tuple arrival rate.

The chart in figure 3(a) shows a breakdown of the traffic in terms of tuples per feed. We aggregated the count of tuples on 1-minute windows. All the charts in this chapter were generated by a streaming-data querying system called Metis, which is being developed at IBM T.J. Watson. Briefly, Metis implements the architecture we have described earlier and can execute dataflows of basic operations over streaming data.

Metis accepts as input the actual packets flowing through the network. For our experiments, we deployed a second machine to re-inject the packets into the network that were read from our trace file. Thus, we reproduced the exact arrival interval recorded previously.

The chart shows a better picture of the NASDAQ feeds' combined traffic profile. The overall traffic is heavily influenced by the most active feeds, UQDF and Prime, and may achieve a rate of over 500,000 tuples per minute during spikes. We observe two kinds of spikes in the curves. One is a local surge on the traffic that doesn't last more than a few minutes at a time. Several of those spikes can be spotted. The second kind of spike reflects a change in the overall volume of tuples. We observe one such spike at the right-hand end of the chart. In figure 3(b) we show the tuple rates of UTDF. We continue our analysis by drilling down into this feed. The charts in figure 4 show the results.

Spikes at this level behave differently. They do not necessarily occur at the same moment in distinct streams. The curves also show that when looked at



**Fig. 3.** Number of parsed tuples of (a) the combined traffic of the most voluminous feeds over a 1-minute sliding window, (b) of the UTDF feed over a 5-seconds one.

closely, collections of trades tend to occur in shortly-spaced bursts. All three curves present a noticeable elevation at the right-hand side of the chart.

In summary, we note that handling the real feeds involves solving the problems of capturing and formatting data fast enough.

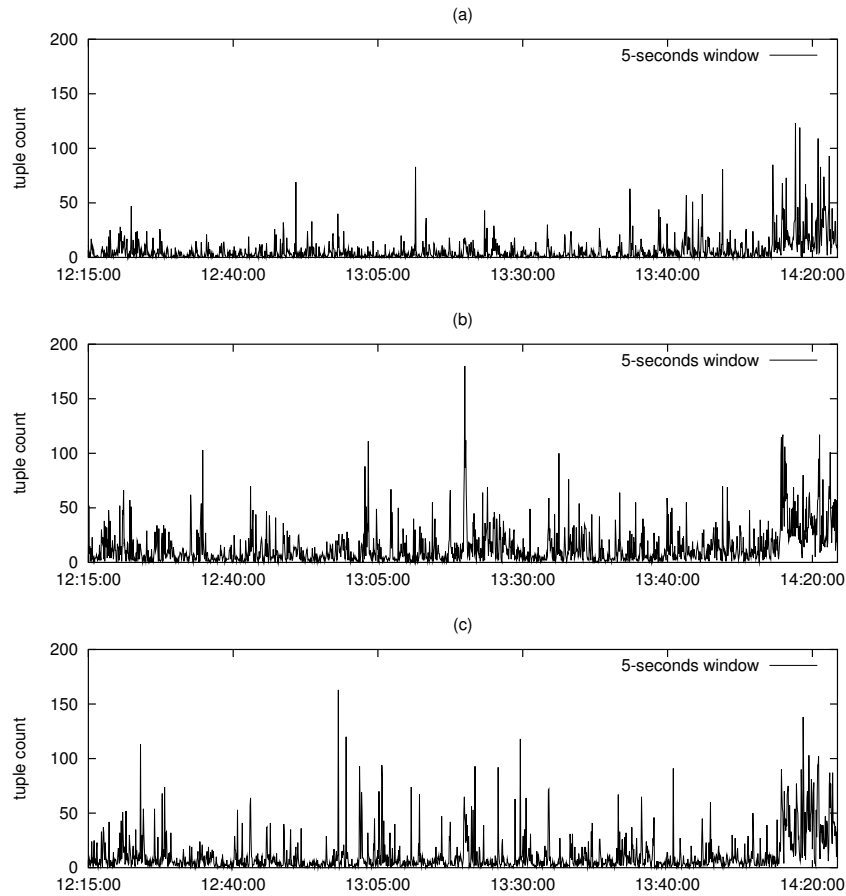
### 3 Architecture

We hope the reader now understand how complex and bursty the NASDAQ feeds are. We believe this is typical. It is now time to describe the components of our architecture 1 in detail.

#### 3.1 Packet Capturing

It is tempting to define an architecture that assumes that data to be processed is necessarily sent to the DSMS machine. Not only may this not always be the case, but also it may be more efficient to copy the packets into the machine's memory in a different way.

In our architecture data is acquired from the network through packet sniffing facilities. The idea is that instead of opening a socket for every single address from which data could be sent to (if that were possible), we capture all packets right from the network interface regardless of their destination.



**Fig. 4.** Number of parsed tuples over a 5-second sliding window of (a) Dell stock trades, (b) Intel stock trades, and (c) Microsoft stock trades.

The facility that makes it possible is called the *Packet Capture Library*, or pcap [3]. The pcap allows one to capture even packets that are not directly addressed to the machine. Other successful streaming data systems have been built using this approach [4].

The pcap accepts filters to be evaluated on each incoming packet which are useful for query processing. For instance, if we have only queries running say on streams generated by the UTDF feed, then we may filter out all other packets.

### 3.2 Classifying

The second component of the architecture routes packets to streams. This is required because even though one may be capturing relevant packets only, there is still the need to separate the streams, or demultiplex the incoming flow.

In the NASDAQ feeds case, this step simply involves building a hash table on the destination address of the packets. Each feed has a fixed and known a-priori number of destination addresses to choose from. In the general case, though, a packet may be routed based on several attributes.

### 3.3 Parsing

Once the packet's stream is identified, its payload can be parsed appropriately. We assume that a packet may carry several records separated by special characters as happens in the NASDAQ feeds. It remains though to identify the type of each of the records.

We use the concept of record *discriminants*. A discriminant is a tuple containing a literal and an offset address within a record. Upon finding a record delimiter, the parser tests for matching discriminants. Once the correct record discriminant is matched, the record can be parsed. The parser for a particular feed is then a collection of discriminants along with the description of their corresponding record layouts.

The layout description typically will map the record's attributes that will become a stream's attributes. For instance, a stream of trades could be generated from the UTDF feed's ST records by using the attributes symbol, price, and volume. We use such a stream in section 4. Attributes of the original record that were not mapped would be discarded.

### 3.4 Taking a new query

A query can act only on streams that have been previously declared to the system. We view a NASDAQ feed as a family of streams, one for each type of record it carries. To describe a feed, one has to enter the following information:

- the network characteristics of the feed (addresses and ports to which it is multicast);
- the layout of the records the stream carries
- for each record layout, information about how to determine the record type (discriminant)

Upon receipt of a query, the system tests whether the needed streams are already being handled. If this is the first query to use a stream *s*, the system then adds information about *s* to the classifier, adds parsing information to the parser, and finally expands the filters to capture that address.

## 4 A Query over an Example Stream

Once the data is formatted into convenient streams, a query over streaming data can be executed by applying a composition operators over the generated stream queue. An operator in this context would be any function that would read its input from this stream's queue and write its output to other streams' queues. We will introduce operators as they appear in our example query.

The query we will present here uses a stream of trades extracted from the UTDF feed. Its layout is Trades(issue symbol, price, volume, date, time). The query does not have to specify any detail about the feed itself. Based on previous information, the system knows how to distill the trades stream from the UTDF feed.

Our query helps a trader execute a large selling order. Traders often need to break a big order into multiple small transactions for the purpose of obtaining the best price. (Large volumes negatively influence the price.) The performance metric they usually have to comply with is the volume-weighted average price (VWAP). For a sequence of trades  $t_0, \dots, t_n$ , VWAP at the trade  $t_n$  is computed as  $(\sum_{i=0}^n t_i.price * t_i.volume) / \sum_{i=0}^n t_i.volume$ . Selling above VWAP is good, and so is buying below it. The challenge here is two-fold: to decide when to trigger the transactions so as to take advantage of some current price momentum, and to do so in doses that do not reverse that trend.

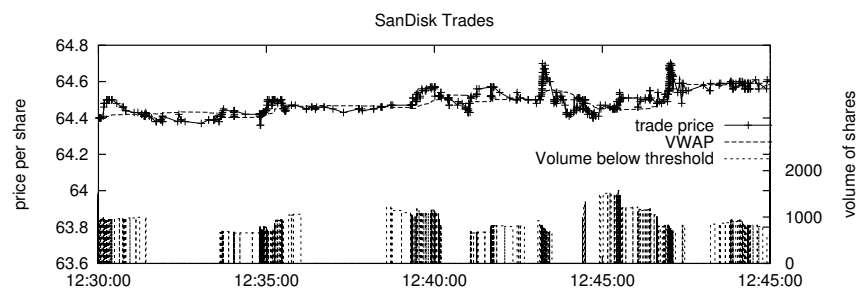
Finding the VWAP involves processing the trades stream. First, a projection operator executes the expression 'price \* volume', thus creating an output queue with this additional column. Then two moving sums (i.e. a cumulative sum over a fixed-sized sliding window) are computed sequentially: the moving sum of 'price \* volume' and that of volume alone. New columns hold these values. Another projection is finally used to compute the division between the two moving sums, which yields the instantaneous VWAP. This query is illustrated in figure 5. We used SanDisk's trades and we can see that the VWAP and the price curves on the figure. A sale window opens when they cross favorably.

Once the window is favorable there is still the issue of deciding how much to sell so not to call attention to the broader sale. This query further computes how many shares could be sold while staying within a low threshold (5%) of the moving sum of the total volume. Figure 5 represents these sale windows with bars indicating the amount of stocks that could be sold.

## 5 Conclusion

Streaming data may be formatted and transmitted in rather complex ways. This chapter shows an architecture for converting complex, burst real streams into a flow of uniform records convenient to a query.

We used, as a running example, NASDAQ feeds which represent a challenge less in the number of packet transmitted per time than in the fact that



**Fig. 5.** An example of a query run over the generated trades stream. VWAP is the volume-weighted average price. This chart shows the selling opportunities times and suggested volumes whenever VWAP is below the current price of the stock.

full packet contents have to be processed. Such feeds can produce peaks of 500,000 tuples per minute.

Our architecture is successful in shielding the feed complexity queries such as the VWAP query. That query performed an analysis over the generated trades stream so to uncover selling windows of opportunity.

Further work on the architecture is to establish a high-level user language for declaratively describing the transformation from raw feeds into streams. This Data Definition Language should ideally be flexible enough to cross domains from finance to network analysis to general sensors.

Other future work concerns the discovery of optimizations and parallelization techniques that could be applied to the architecture. Each of the components deserves to be studied separately. Data capturing is critical because it is a real-time activity. Failing to check for packets frequently enough may cause packet loss. Classifying and parsing also have performance constraints and are best met if they could be done in constant time regardless of the number of streams being handled.

## Acknowledgment

We would like to thank Ed Bierly and Steve Miano for kindly providing us with the NASDAQ feeds' dump used here and Ted Johnson for several comments on early drafts of this paper.

## References

1. Market Data Distribution, NASDAQ Trader site, <http://www.nasdaqtrader.com>
2. The UTP Plan Trade Data Feed Specification, NASDAQ Market Data Distribution, available at <http://www.nasdaqtrader.com/trader/mds/utpfeeds/utpfeeds.stm>

3. Tcpdump and Libpcap site, <http://www.tcpdump.org>
4. The Gigascope Stream Database (2003), Charles D. Cranor, Theodore Johnson, Oliver Spatcheck, and Vladislav Shkapenyuk, IEEE Data Eng. Bulletin, 26(1):27–32.