NEXMark – A Benchmark for Queries over Data Streams
DRAFT

Pete Tucker, Kristin Tufte, Vassilis Papadimos, David Maier

OGI School of Science & Engineering at OHSU
{ptucker,tufte,vpapad,maier}@cse.ogi.edu

1 Introduction

A lot of research has focused recently on executing queries over data streams. This recent attention is due to the large number of data streams available, and the desire to get answers to queries on these streams in real time. There are many sources of data streams: environmental sensor networks, network routing traces, financial transactions and cell phone call records. Many systems are currently under development to execute queries over data streams [BW01, CCC+02, MF02, NDM+00, SH98]. Further, many ideas have been presented to improve the execution of queries over data streams [ABB+02, MSHR02, TMSF02]. It is important to be able to measure the effectiveness of this work. To this end, we present the Niagara Extension to XMark benchmark (NEXMark).

This is a work in progress. We are circulating it now for feedback. We have three goals: To define a benchmark, to provide stream generators, and to define metrics for measuring queries over continuous data streams.

The XMark benchmark [SWK+01] is designed to measure the performance of XML repositories. The benchmark provides a data generator that models the state of an auction in XML format, and various queries over the generated data. An abbreviated schema for XMark is shown in Figure 1.

2 Adapting XMark to Streaming Data

The XMark scenario can realistically be extended to an on-line auction system such as EBay [EBA] using data streams. In an on-line auction, hundreds of auctions for individual items are open at any given time. New people are continuously registering with the system. New items are continuously submitted for auction. Bids are continuously arriving for items. Based on these continuous actions, it is easy to imagine three kinds of data streams passing data into the main auction system. In the NEXMark scenario, we use three kinds of business objects. Each kind of business object represents different interactions users have with the auction, as follows:

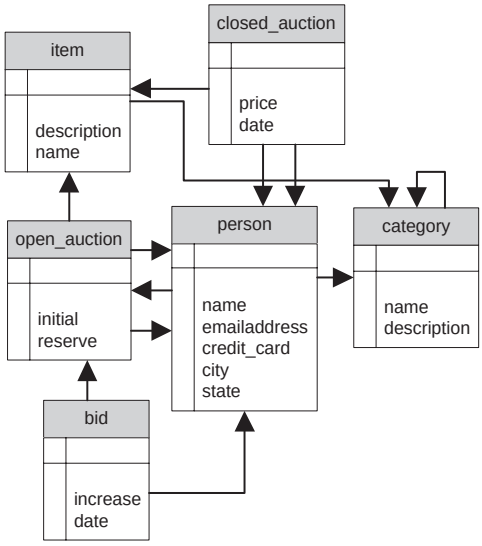

Figure 1: Abbreviated Conceptual Schema for XMark Auction Data

- All users register with the `Person` business object in order to participate in auctions as buyers or sellers. The `Person` business object streams the registration information into the main auction system.
- Sellers submit items they want to sell to the `Auction` business object. Information about the item includes description, reserve, start and end time for the auction, and quantity. The description and quantity of the item is streamed to the main system as a `item` entity. The reserve and start and end time are streamed to the main system as `open_auction` entity to begin the auction. When the auction expires, the `Auction` business object signals the close of an auction to the main system using a `close_auction` entity.

- Buyers enter bids for existing auctions to the `Bid` business object. Each bid is streamed from the business object into the main system. There may be more than one `Bid` business object deployed, depending on the scale of the test. Scaling is discussed further in Section 4.2.

Additionally, a static file is stored on disk. This file contains data, such as category information, which changes infrequently and is not presented as a stream. The NEXMark architecture is shown in Figure 2.

There are many interesting queries that users would want to execute in this kind of system. These queries might be over data from single business objects, over joins or unions over data from multiple business objects, and over joins over data from business objects with stored data.

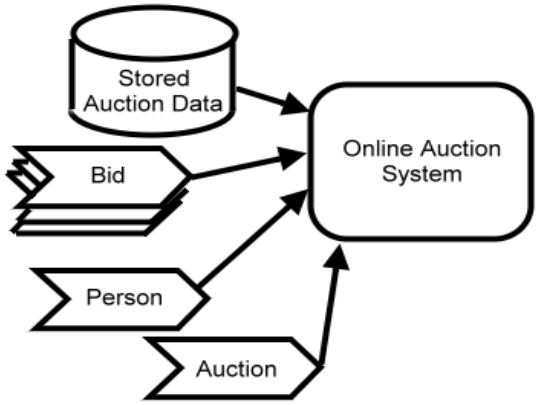

Figure 2: Architecture for the On-line Auction System

Though the NEXMark benchmark extends XMark, we do not intend for NEXMark to be strictly for XML data streams. It can be implemented using data in any format. A simplified version of the schema for the NEXMark benchmark is shown in Figure 3. The `category` entity allows sellers to categorize their items in the auction hierarchically. Users can then set up watch lists for new auctions on items in specific categories.

We need to make some changes in the XMark schema to facilitate the new streams and business objects. These changes are listed below:

- XMark represents bid prices as increases over the previous price. This representation is easy to generate, but not realistic if bids are arriving in different streams from different servers. Therefore, the `Bid` business object will generate `bid` entities which include the actual bid price.

- The `closed_auction` entity comes from the `Auction` business object. The `Auction` business object does not know the closing price for the item in the auction, so it is not included. The `closed_item` entity only contains the closing date and buyerid for the auction.

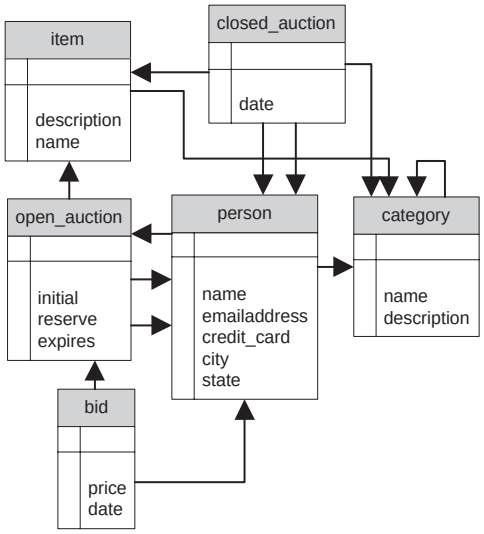

Figure 3: Abbreviated Schema for NEXMark Auction Data

3 Queries

The NEXMark benchmark addresses what we call *Stream-In, Stream-Out* queries. Stream-In, Stream-Out queries are continuous queries that take as input a stream, or streams, of data, perform operations on the stream(s) and produce a stream of data as output. An example simple Stream-In, Stream-Out query is: given an input stream of bids, output all bids on a particular set of items.

Two interesting classes of queries over streams that are not addressed by this benchmark are triggers and ad-hoc queries. Triggers differ from Stream-In, Stream-Out queries in that triggers perform an action based on the results of a query, while Stream-In, Stream-Out queries only produce results. In addition, we envision that Stream-In, Stream-Out queries will have a more continuous output stream than traditional trigger queries, which may fire only irregularly. Ad-hoc queries over streams are one-time queries which are answered based on the "current state" of the stream. An example ad-hoc query is: get the current price for the item with id 5120. While both of these classes of queries are important, in order to keep the benchmark focused, we do not include these types of queries. We proceed to describe the queries in the benchmark.

3.1 Queries in NEXMark

This section describes the queries that make up the NEXMark benchmark. We express the queries in SQL, since SQL has been widely adopted; however SQL is not a requirement for the system. There are eight

queries in the benchmark, the second four are window queries.

Before we describe the queries in the benchmark, we make a note on benchmark scaling. As described in Section 4.2, larger scale NEXMark implementations will be required to process multiple bid streams. For simplicity, all the queries are expressed using a single bid stream (relation). For higher-scale implementations, this single bid stream (relation) is presumed to be the union of the multiple input bid streams.

3.2 Query 1- Query 4

Query 1 *Currency Conversion*

The purpose of Query 1 is to test the processing speed of the stream system and to provide a reference point for the rest of the queries. One essential feature tested by Query 1 is parse speed. Parse speed is particularly important in stream systems because stream systems operate on data that is not in native database format and reading such data can be slow. Query 1 takes an incoming bid stream and converts the prices of the bids from U.S. dollars to Euros.

```
SELECT itemid, DOLTOEUR(price),
  bidderId, bidTime
FROM bid;
```

DOLTOEUR is a function which takes a price in dollars and converts it to Euros.

Query 2 *Selection*

Query 2 selects all bids on a set of five items and tests the stream system's selection operation.

```
SELECT itemid, price
FROM bid
WHERE itemid = 1007 OR
  itemid = 1020 OR
  itemid = 2001 OR
  itemid = 2019 OR
  itemid = 1087;
```

Query 3 *Local Item Suggestion*

Query 3 is designed to test join functionality. We imagine that buyers want to find items in a particular category that are for sale by sellers who live near them. Query 3 performs a join between the stream of new items for auction and the people registered with the auction system. The query should output a result every time a new item becomes for sale in category 10 in Oregon. Note that the result should not contain items that are no longer up for auction (closed auctions).

```
SELECT person.name, person.city,
  person.state, open_auction.id
FROM open_auction, person, item
WHERE open_auction.sellerId = person.id
  AND person.state = 'OR'
  AND open_auction.itemid = item.id
  AND item.categoryId = 10;
```

Query 4 *Average Price for a Category*

Users who are considering selling an item want to know the average closing price for other items in that category. The category data is stored in a static file known by the system. Query 4 joins the category file with the closed_auction stream to calculate average closing price for each. The query should output updated prices when new closing prices arrive for a particular group.

```
SELECT C.id, AVG(CA.price)
FROM category C, item I, closed_auction CA
WHERE C.id = I.categoryId
  AND I.id = CA.itemid
GROUP BY C.id;
```

3.3 Query 5 - Query 8 - Window Queries

The last four queries in the benchmark are window queries; three are window group-bys and one is a window join. We have taken care to use various types of windows in the benchmark including sliding and fixed windows and time-based (logical) and event-based (physical) windows. The windows are specified using SQL-99 style specifications as proposed by Babcock *et al.* [BBD+02]. The keyword RANGE is used to express time-based (logical) windows - for example, the average price of all bids in the last ten minutes. The keyword ROWS is used to express event-based (physical) windows - for example, the average price of the last ten bids.

Query 5 *Hot Items*

This query selects the item with the most bids in the past one hour time period; the "hottest" item. The results are output every minute. This query uses a time-based, sliding window group by operation.

```
SELECT bid.itemid
FROM bid [RANGE 60 MINUTES PRECEDING]
WHERE (SELECT COUNT(bid.itemid)
  FROM bid [PARTITION BY bid.itemid
    RANGE 60 MINUTES PRECEDING])
  >= ALL (SELECT COUNT(bid.itemid)
      FROM bid [PARTITION BY bid.itemid
        RANGE 60 MINUTES PRECEDING];
```

Please correct us if we have this syntax wrong.

Query 6 *Average Selling Price by Seller*

Query 6 calculates, for each seller, the average selling price of items sold by that seller. For example, auction site administrators may be interested in knowing which users sell the highest price items. This query uses an event-based, sliding window group by.

```
SELECT AVG(CA.price), CA.sellerId
FROM closed_auction CA
  [PARTITION BY CA.sellerId
  ROWS 10 PRECEDING];
```

Query 7 *Highest Bid*

Query 7 monitors the highest price items currently on auction. Every ten minutes, this query returns the highest bid (and associated itemid) in the most recent ten minutes. This query uses a time-based, fixed-window group by. The syntax FIXEDRANGE is used in place of RANGE to indicate that the highest bid should be evaluated every ten minutes instead of over a sliding ten minute window.

```
SELECT bid.price, bid.itemid
FROM bid where bid.price =
  (SELECT MAX(bid.price)
  FROM bid [FIXEDRANGE
    10 MINUTES PRECEDING]);
```

Query 8 *Monitor New Users*

This query finds people who put something up for sale within twelve hours of registering to use the auction service. This query could be used to track new users for user followup or to make sure the new users are "behaving". This query uses a sliding window join over a logical or time-based window.

```
SELECT person.id, person.name
FROM person [RANGE 12 HOURS PRECEDING],
  open_auction [RANGE 12 HOURS PRECEDING]
WHERE person.id = open_auction.sellerId;
```

4 NEXMark Implementation

4.1 Firehose Stream Generator

We are in the process of developing the Firehose Stream Generator (FSG) to simulate the output from the business objects required for this benchmark. The FSG will be configurable to output new persons, bids for items, and auctions that have opened and closed. Additionally, the firehose will be configurable to output data at specific stream rates. The number of streams required and the stream rate is determined by the testing scale, described below.

4.2 Scale

It is desirable to allow for different levels of scaling in the system. Scaling in the NEXMark system is based on the size of the checkpoint file, the number of bid streams, the rate of the bid streams, and the duration of the test. We will define three scale levels: small, medium, and large.

- **Static file size** The static file contains static data that will to be queried along with data from the streams. The static file contains data that is permanent or at least semi-permanent such as the category list. A larger file may take longer to query against, making it more difficult to handle faster input stream rates.
- **Number of bid streams** As the system gets larger, there will be more items to bid on, and more bids coming into the system. Large-scale systems must execute over more bid streams than small-scale systems.
- **Stream Rate** The rate of the item stream is one tenth the rate of the bid stream, and the rate of the person stream is one tenth the rate of the item stream (one one-hundredth the rate of the bid stream). The stream rate for large-scale systems will be faster than that of small-scale systems.
- **Test duration** This benchmark is intended for systems that execute queries over continuous data streams. However, in order to accurately compare the test results for many systems, we define a duration for the execution of the test. Large-scale systems will be required to run longer than small-scale systems.

5 Metrics

Measuring a system that queries continuous data streams is not a straightforward task. Traditionally, DBMS benchmarks measure the length of time to execute a query. We cannot apply this scheme to data stream query systems, since the input is continuous and theoretically the queries never end.

The most important things to measure about a stream system are: how fast can the system process data? and how accurate are the results? The accuracy measurement is important because as the input rate increases, a system may have to drop tuples or take other measures to keep up with the input. This behavior is acceptable, and will result in approximate

answers. It is important to measure just how accurate the approximate answers are.

With these ideas in mind, we propose two metrics: Input Stream Rate and Output Matching. These two values should be reported together. For example, one would say System X ran at Y accuracy at Input Stream Rate of Z MB/sec. Input Stream Rate is straightforward; below we discuss Output Matching further. Finally, we touch on Tuple Latency which is the metric we are using in our current experiments.

5.1 Output Matching

Output matching is a combined metric of output timeliness and accuracy in Stream-in, Stream-out queries. For an example, we use the following query which outputs the current price of a particular item for auction.

```
SELECT bid.price
FROM bid
WHERE bid.itemid = 5192;
```

Used as a Stream-in, Stream-out query, this query outputs a new price each time a new bid on item 5192 arrives. We assume, only for this example, that bids come in order of increasing price. Thus the output stream will be a series of increasing prices for item 5192. This is a very simple query, but serves well as an illustration.

In this example query, the input is a stream of bid prices $b_1, b_2, \ldots, b_n$ for some item, coming in at times $t_1, t_2, \ldots, t_n$. To implement Output Matching, we assume an ideal stream database which operates with infinite speed and accuracy. This ideal system outputs an updated price for the item instantly when the new bid arrives. Any actual implementation, however, will introduce some processing delay d_i, and will output the price corresponding to bid increase b_i at time $t_i + d_i$.

We can plot a graph of the output values vs. time for the ideal system and for an implementation. Figure 4 shows such a graph. In this graph, the ideal system is represented by the solid line, the implementation by the dotted line. The Output Matching metric is the value of the area between the two lines (shown as a shaded area in Figure 4) normalized by time. This shaded area can be interpreted as an indication of the timeliness and accuracy of the results. The slower and less accurate the implementation is, the more space there will be between the solid and dotted lines and the higher the value of the Output Matching metric, as desired.

We feel the Output Matching metric is a good method for measuring the timeliness and accuracy of a stream system. It is clear that the Output Matching Metric can be used for queries, such as the example query, which have a numerical result. We can generalize output matching to cover not just queries with ordered output domains (such as money, in this example), but most stream-in, stream-out queries with a notion of a "current result". All that is required is a function which given two possible query results returns a numeric value representing how "different" those results are. In the worst case, this degrades to a 0 if the two results are the same and 1 if they are different. However, typically a more satisfying function can be used.

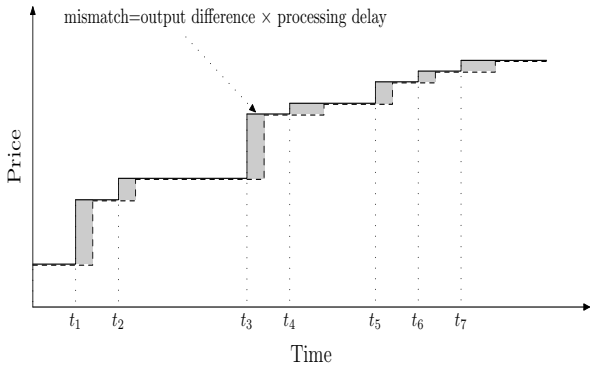

Figure 4: Output Matching for Example Query

5.2 Tuple Latency

In our current experiments, we use the Tuple Latency metric. Tuple Latency does not take into account approximate results and is therefore not as good as Output Matching, but it is easier to measure and provides a good starting point. Tuple Latency is the measure of how long it takes for a tuple that is relevant to the result to go through the system.

6 Conclusion

There are a number of systems being developed that execute queries over continuous data streams. We propose an extension to the XMark benchmark, called NEXMark, to measure the effectiveness of these stream querying systems. The extension is straightforward and realistic. We define queries, and provide a source for the data streams and a client to accept output from the system and report benchmark results.

We are trying to represent applications that execute queries over streams of event data with this benchmark. These applications execute filter queries, aggregate queries, and join queries over incoming streams as well as stored data.

References

[ABB+02] Arvind Arasu, Brian Babcock, Shivnath Babu, Jon McAlister, and Jennifer Widom. Characterizing memory requirements for queries over continuous data streams. In *Proceedings of the 21st*

ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS), pages 221–232, June 2002.

[BBD+02] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and Issues in Data Stream Systems. In *Proceedings of the 21st ACM Symposium on Principles of Database Systems (PODS 2002)*, June 2002, Madison, WI.

[BW01] Shivnath Babu and Jennifer Widom. Continuous queries over data streams. *SIGMOD Record*, 30(3), September 2001.

[CCC+02] Don Carney, Ugur Cetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Monitoring streams – a new class of data management applications. In *Proceedings of the 28th Conference on Very Large Data Bases*, August 2002.

[EBA] eBay home page. http://www.ebay.com/.

[MF02] Samuel Madden and Michael J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *Proceedings of the 18th International Conference on Data Engineering*, pages 555–566, February 2002.

[MSHR02] Samuel Madden, Mehul Shah, Joseph M. Hellerstein, and Vijayshankar Raman. Continuously adaptive continuous queries over streams. In *Proceedings of the ACM Special Interest Group on Management of Data*, pages 49–60, June 2002.

[NDM+00] Jeffrey Naughton, David DeWitt, David Maier, Jianjun Chen, Leonidas Galanis, Kristin Tufte, Jaewoo Kang, Qiong Luo, Naveen Prakash, and Feng Tian. The Niagara query system. Technical report, University of Wisconsin, March 2000.

[SH98] Mark Sullivan and Andrew Heybey. Tribeca: A system for managing large databases of network traffic. In *Proceedings of the 1998 USENIX Annual Techincal Conference*, June 1998.

[SWK+01] Albrecht Schmidt, Florian Waas, Martin Kersten, Daniela Florescu, Ioana Manolescu, Michael J. Carey, and Ralph Busse. The XML benchmark project. Technical Report INS-R0103, Centrum voor Wiskunde en Informatica, April 2001. http://monetdb.cwi.nl/xml.

[TMSF02] Pete Tucker, David Maier, Tim Sheard, and Leonidas Fegaras. Enhancing relational operators for querying over punctuated data streams. *manuscript*, 2002. URL: http://www.cse.ogi.edu/dot/niagara/pstream/punctuating.pdf.