

CS848 Presentation Report

Execution of Streaming Operators

Jingchi (Evan) Chen
E-mail: j34chen@cs.uwaterloo.ca

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada

Feb 14, 2006

1. Introduction.

As database system becomes more and more complex, we not only have static, persistent data which are stored on disks, we also have dynamic, ongoing data which depend on timestamp. These data models are called data sequences and data streams. The definition is given below.

Definition 1: *Sequence: Let t_1, \dots, t_n be tuples from a relation R . Then, the list $S = [t_1, \dots, t_n]$ is called a sequence, of length n , of tuples from R . The empty sequence is denoted by $[\]$; $[\]$ has length 0.*

Data streams are basically unbounded sequences, in other words, infinite data sequences.

The intuitive question that might come to our mind is, can we use the query language which is used for persistent data, such as SQL to query data stream? Short answer: No!

Since some queries in SQL need to know the entire input before they can produce the output, and now we are not dealing with persistent data, we are dealing with infinite data sequences. Some sort of continuous queries are needed to query data stream, and this is one of the severe limitations SQL has since SQL suffers from additional expressive power problems.

We can extend SQL to allow such continuous queries for data streams. We can begin by focusing on nonblocking queries (NB) which can be supported on data streams.

2. Nonblocking query operators

What are nonblocking query operator and blocking query operator, what is the difference between those two, and how do they function differently with data streams. You will know the answers to these questions in this section.

Definition 2: *A blocking query operator is a query operator that is unable to produce the first tuple of the output until it has seen the entire input.*

Definition 3: *A nonblocking query operator is one that produces all the tuples of the output before it has detected the end of the input.*

As we can see from the definitions, only nonblocking query operators are suitable for data streams since continuous queries must return answers without waiting for the entire input to be finished, which is impossible for data streams. Also we may realize the reason why SQL has a severe limitation in query data streams. Most of the traditional aggregates or constructs in SQL such as *NOT IN*, *NOT EXISTS*, *ALL* and *EXCEPT*, they will never produce outputs until they see the entire input.

So what qualifies an operator to be a nonblocking operator?

Definition 4: A non-null operator G is said to be

--**blocking**, when for every sequence S of length n , $G_j(S) = []$ for every $j < n$, and $G_n(S) = G(S)$

--**nonblocking**, when for every sequence S of length n , $G_j(S) = G(S_j)$, for every $j \leq n$.

This paper also proved that all monotonic queries, and only those, can be expressed using nonblocking computations. The theorem is as follows.

Proposition 1 A function $F(S)$ on a sequence S can be computed using a nonblocking operator, iff F is monotonic with respect to the partial ordering \leq .

For more details on the proof of this theorem, please consult the paper.

3. User-Defined Aggregates

Now, Let us get back to where we started, how do we extend SQL to make it qualify querying data streams? One possible way is to define application specific aggregates which have the same functionality as traditional aggregates in SQL. In this case, we can go around the limitations of SQL by making user-defined aggregates (UDA) which “mimic” the standard aggregates in SQL.

Thus UDAs operate as general stream transformers, and they are used in the same way as traditional built-in aggregates in SQL.

Here are 2 examples of UDAs which define the standard AVG in SQL. The first one is blocking and the second one is nonblocking.

Example 1:

```
AGGREGATE myavg(Next Int) : Real
{
  TABLE state(tsum Int, cnt Int);
  INITIALIZE : {
    INSERT INTO state VALUES (Next, 1);
  }
  ITERATE : {
    UPDATE state
      SET tsum=tsum+Next, cnt=cnt+1;
  }
  TERMINATE : {
    INSERT INTO RETURN
      SELECT tsum/cnt FROM state;
  }
}
```

Example 2:

```

AGGREGATE tumble_avg(Next Int) : Real
{
  TABLE state(tsum Int, cnt Int);
  INITIALIZE : {
    INSERT INTO state VALUES (Next, 1)}
  ITERATE: {
    UPDATE state
      SET tsum=tsum+Next, cnt=cnt+1;
    INSERT INTO RETURN
      SELECT tsum/cnt FROM state
      WHERE cnt % 200 = 0;
    UPDATE state SET tsum=0, cnt=0
      WHERE cnt % 200 = 0
  }
  TERMINATE : { }
}

```

As we have observed here, not every UDA is blocking. One easy way to identify the nonblocking UDA is that its *TERMINATE* clause is either empty or absent. Therefore, nonblocking UDAs play a very important role in data sequence and data stream applications. Unlike traditional aggregates, they are well designed to suit state-based reasoning and queries.

4. Completeness on DB relations.

Speaking of the ultimate state machine: Turing Machine, SQL is not Turing complete since it can not express all query applications. On the other hand, it is shown in the paper that UDAs can compute any arbitrary queries encoded in Turing Machine. In the implementation presented by the paper, a user can define the Turing machine by 4 elements:

```

--A transition map(E1)
--Accepting states(E2)
--A tape containing the input(E3)
--An initial state(E4).

```

And we have the following mappings from UDA to the Turing machine:

```

E1 --- a table called transition
E2 --- a table called accept
E3 --- a table called tape
E4 together with the leftmost symbol --- a table called current

```

From the “finding the maximum” example in this paper, we have realized that although UDAs can express any input functions encoded in Turing machine, some of the UDAs are blocking, which might not be suitable for continuous queries for data streams and data sequences.

5. Completeness on Data Streams.

The goal for a valid operator for a data stream is “queries over streams run continuously over a period of time and incrementally return new results as new data arrive”.

As we proposed before, any monotonic function can be computed using nonblocking query operators. So how do we compute monotonic functions on data streams using UDAs? It can be divided into 2 cases: a single data stream and multiple data streams.

For a single data stream, it is easy. We can use 3 local tables called *IN*, *TAPE* and *OUT*, and follow the following steps:

1. *Append the encoded new tuple to IN*
2. *Copy IN to TAPE, and compute F(IN) - OUT*
3. *Return the result obtained in 2 and append it to OUT.*

For multiple data streams, it is a bit more complicated. When new tuples come, we need to merge them into one single stream before we can apply any UDAs. To achieve the goal of merge multiple incoming data streams, we will explicitly use timestamped data streams, where tuples are ordered by the increasing values of their timestamps.

Another theorem follows given the methods on calculating monotonic functions on data streams.

Proposition 2: *NB-Completeness. Every computable monotonic function on timestamped data streams can be expressed using NB-UDAs and union.*

We can demonstrate this theorem by an example as follows.

Given two data streams of phone call records:

```
StartCall(callID, time);  
Endcall(callID, time);
```

And we are interested in calculating the length of each phone call in an ongoing basis. First we can union these two data streams into a new table called *CallRecords* sorted by the arrival timestamp.

```
SELECT callID, length(time, tag) AS CallLength,  
FROM  
  (SELECT callID, time, 'start'  
   FROM StartCall  
   UNION ALL  
   SELECT callID, time, 'end'  
   FROM EndCall) AS  
CallRecord (callID, time, tag)
```

GROUP BY callID;

Then we can use UDA length to compute the actual length of each phone call.

AGGREGATE length(time, tag) : (CallLength)

```
{ TABLE state(ttime);
  INITIALIZE: ITERATE :{
    INSERT INTO state VALUES(time);
    INSERT INTO RETURN
      SELECT time-ttime FROM state
      WHERE tag='end';
    INSERT INTO RETURN
      SELECT ttime-time FROM state
      WHERE tag='start';
  }
}
```

6. Conclusion.

The existence of data sequences and data streams needs some significant changes to traditional database technology. This paper focused on the changes of query languages. We are able to find out the limitations of traditional SQL when applying to data streams because data streams require nonblocking queries.

Thus we have proved that SQL is no longer complete for nonblocking queries. To solve this problem when dealing with data streams, we introduced an extension of SQL: User-defined Aggregates (UDA). Now we can make SQL Turing complete with the extension functionality of UDA.

In the end, we also proved that any query functions that support UDA and nonblocking *UNION* are NB-complete.