Exploiting Punctuation Semantics in Data Streams

Pete Tucker, David Maier
Oregon Graduate Institute at OHSU
20000 Walker Road
Beaverton, OR 97006
{ptucker,maier}@cse.ogi.edu

Applications that process data streams are becoming common: financial applications process streams of stock ticker data; telephone network monitoring applications process streams of call data. These applications often are queries over streams, so it seems natural to use a database management system instead of a custom application. However, some traditional relational operators are not conducive to stream processing. *Blocking operators*, such as `sort`, wait until the entire input has been read. *Unbounded stateful operators*, such as `duplicate elimination`, maintain state that can grow without bound.

For example, consider a warehouse that contains temperature-sensitive items. Sensors are scattered throughout the warehouse, reporting the temperature every thirty seconds. A main system reads the data from the sensors, and reports every hour the maximum temperature at any of the sensors. Clearly, this can be solved using a query that unions the sensor inputs, groups the data for each hour, then outputs the maximum temperature by hour, as in Figure 1. Unfortunately, the `group-by` operator is a blocking operator, so the system will never produce an output.

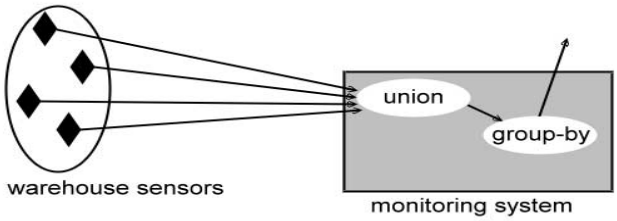

Figure 1. A sensor system using a DBMS.

We propose embedding *punctuations* into data streams. A punctuation is a predicate that describes a subset of tuples. It informs a stream processor that no tuples exist after that punctuation that satisfy its predicate. A conceptual view of a punctuated stream is shown in Figure 2.

Two functions are required for processing punctuations: `match` and `combine`. The `match` function takes a tuple and a punctuation and returns `true` if that tuple satisfies the predicate described by the punctuation. The `combine` function takes two punctuations, and returns a single punctuation that is the logical intersection of the two inputs. That is, `match (t, (combine (`p_1`,`p_2`)))` $\Rightarrow$ `match (t,`p_1`)` $\wedge$ `match (t,`p_2`)`.

Figure 2. A punctuated data stream. Items with a 'P' are punctuations.

We define three kinds of rules that describe how operators process punctuations: *Pass rules* describe what results an operator can output early. *Purge rules* describe what state an operator can purge. *Propagation rules* describe what punctuation an operator should output. Each operator has its own specific set of these rules.

We can define a pass rule for `group-by` to unblock it; groups can be output that match punctuations that describe only the grouping attribute(s). In the warehouse example, if the operator receives a punctuation that describes a particular hour, results for that hour are output.

We can define a purge rule for `group-by` to decrease its state. The purge rule for `group-by` is its pass rule. That is, groups that are output are removed from state.

Propagation rules for some operators, such as `union`, are not trivial. If a punctuation arrives from one of its inputs, tuples matching that punctuation can still arrive on another input. Therefore, the propagation rule for `union` is to output only combinations of punctuations from *all* inputs using the `combine` function. In the warehouse example, a single punctuation for a particular hour is output when all inputs emit a punctuation for that hour.

Funding provided by DARPA through NAVY/SPAWAR Contract No. N66001-99-1-8908, NSF ITR award IIS0086002, and NSF grant IIS-9811525. For more information, visit http://www.cse.ogi.edu/~ptucker/PStream.