C

Continuous Queries

Martin Hirzel
IBM Research, Yorktown Heights, NY, USA

Synonyms

Streaming SQL queries; Stream-relational queries; StreamSQL queries

Definition

A continuous query in an SQL-like language is a declarative query on data streams expressed in a query language for streams derived from the SQL for databases.

Overview

Just like data that is stored in a relational database can be queried with SQL, data that travels in a stream can be queried with an SQL-like query language. For databases, the relational model and its language, SQL, have been successful because the relational model is a foundation for clean and rigorous mathematical semantics and because SQL is declarative, specifying what the desired result is without specifying how to compute it (Garcia-Molina et al. 2008). However, the classic relational model assumes that data resides in relations in a database. When data travels in a stream, such as for communications, sensors, automated trading, etc., there is a need for continuous queries. SQL dialects for continuous queries fill this need and inherit the advantages of SQL and the relational model. Furthermore, SQL-like streaming languages capitalize on the familiarity of SQL for developers and of implementation techniques from relational databases.

There are various different SQL-like streaming languages. This article illustrates concepts using CQL (the continuous query language, Arasu et al. (2006)) as a representative example, because it has clean semantics and addresses the interplay between streams and relations. Section "Findings" explores other SQL-like streaming languages beyond CQL.

SQL-Like Syntax

The surface syntax for streaming SQL dialects borrows familiar SQL clauses (such as select, from, and where) and augments them with streaming constructs (which turn streams into relations and vice versa). Consider the following CQL (Arasu et al. 2006) query with extensions adapted from Soulé et al. (2016):

© Springer International Publishing AG, part of Springer Nature 2018
S. Sakr, A. Zomaya (eds.), *Encyclopedia of Big Data Technologies*,
https://doi.org/10.1007/978-3-319-63962-8_305-1

```
1 Quotes  : { ticker : string, ask : int } stream;
2 History  : { ticker : string, low : int } relation;
3 Bargains : { ticker : string, ask : int, low : int } stream
4   = select istream(*)
5     from Quotes[now], History
6     where Quotes.ticker == History.ticker and Quotes.ask <= History.low;
```

Line 1 declares *Quotes* as a stream of {*ticker*, *ask*} tuples, and Line 2 declares *History* as a relation of {*ticker*, *low*} tuples. Neither *Quotes* nor *History* is defined with a query in the example. Lines 3–6 declare *Bargains* and define it with a query. Line 3 declares *Bargains* as a stream of {*ticker*, *ask*, *low*} tuples. Line 4 specifies the output using the **istream** operator, which creates a stream from the insertions to a relation, using * to pick up all available tuple attributes. Line 5 joins two relations, *Quotes*[**now**] and *History*, where *Quotes*[**now**] creates a relation from the current contents of stream *Quotes*. Finally, Line 6 selects only tuples satisfying the given predicate, whereas any tuples for which the predicate is false are dropped.

The above example illustrates the core features of CQL, viz.: using operators such as **now** to turn streams into relations, using SQL to query relations, and using operators such as **istream** to turn relations into streams. For more detail, the following paragraphs explain the CQL grammar, starting with the top-level syntax:

```
program   ::= decl+
decl      ::= ID ':' tupleType declKind ('='query)? ';'
tupleType ::= '{'(ID ':' TYPE)+, '}'
declKind  ::= 'relation '|' stream'
query     ::= select from where? groupBy?
```

A program consists of one or more declarations. Each declaration has an identifier (*ID*), a tuple type (one or more attributes specified by their identifiers and types), a declaration kind (either **relation** or **stream**), and an optional query. The grammar meta-notation contains superscripts for optional items ($X^?$), repetition (X^+), and repetition separated by commas ($X^{+,}$). Finally, a query consists of mandatory select and from clauses and optional where and group-by clauses. Next, we look at the grammar for the select clause, which specifies the query output:

```
select       ::= 'select' outputList |'select' relToStream '(' outputList ')'
relToStream  ::= 'istream' | 'dstream' | 'rstream'
outputList   ::= '*' | projectItem+, | aggrItem+,
projectItem  ::= expr ('as'ID)?
aggrItem     ::= AGGR '('ID*, ')'('as' ID)?
```

A select clause either specifies an output list directly or wraps an output list in a relation-to-stream operator. In the first case, the query output is a relation, while in the second case, the query output is a stream. There are three relation-to-stream operators: **istream** captures insertions, **dstream** captures deletions, and **rstream** captures the entire relation at any given point in time. Here, a relation at a given point in time is a bag of tuples (i.e., an unordered collection of tuples that can contain duplicates). A stream is an unbounded bag of timestamped tuples (pairs of $\langle timestamp, tuple \rangle$). The grammar for *outputList* is borrowed from SQL. Next, we look at the

grammar for the from clause, which specifies the query input:

$$
\begin{array}{lll}
\textit{from} & ::= & \text{`}\textbf{from}\text{'}\ \textit{inputItem}^{+,} \\
\textit{inputItem} & ::= & \textit{ID}\ (\text{`[' }\textit{streamToRel}\ \text{`]'})^{?}\ (\text{`}\textbf{as}\text{'}\ \textit{ID})^{?} \\
\textit{streamToRel} & ::= & \text{`}\textbf{now}\text{'} \mid \text{`}\textbf{unbounded}\text{'} \mid \textit{timeWindow} \mid \textit{countWindow} \\
\textit{timeWindow} & ::= & \textit{partitionBy}^{?}\ \text{`}\textbf{range}\text{'}\ \textit{TIME}\ (\text{`}\textbf{slide}\text{'}\ \textit{TIME})^{?} \\
\textit{countWindow} & ::= & \textit{partitionBy}^{?}\ \text{`}\textbf{rows}\text{'}\ \textit{NAT}\ (\text{`}\textbf{slide}\text{'}\ \textit{NAT})^{?} \\
\textit{partitionBy} & ::= & \text{`}\textbf{partition}\text{'}\ \text{`}\textbf{by}\text{'}\ \textit{ID}^{+,}
\end{array}
$$

An input item either identifies a relation directly or applies a stream-to-relation operator to a stream identifier. Stream-to-relation operators are written postfix in square brackets, reminiscent of indexing or slicing syntax in other programming languages. There are four such operators: **now** (tuples with the current timestamp), **unbounded** (tuples up to the current timestamp), **range** (time-based sliding window), and **rows** (count-based sliding window). Sliding windows can optionally be partitioned, in which case their size is determined separately and independently for each unique combination of the specified partitioning attributes. Sliding windows can optionally specify a slide granularity. Finally, we look at the grammar for *where* and *groupBy* as examples of other classic SQL clauses:

$$
\begin{array}{lll}
\textit{where} & ::= & \text{`}\textbf{where}\text{'}\ \textit{expr} \\
\textit{groupBy} & ::= & \text{`}\textbf{group}\text{'}\ \text{`}\textbf{by}\text{'}\ \textit{ID}^{+,}
\end{array}
$$

The where clause selects tuples using a predicate expression, and the group-by clause specifies the scope for aggregation queries. These clauses in CQL are borrowed unchanged from SQL. For brevity, we omitted other classic SQL constructs, such as distinct, union, having, or other types of joins, which streaming SQL dialects such as CQL can borrow from SQL as is.

Typically, before execution, queries in SQL or its derivatives are first translated into a logical query plan of operators, which is the subject of the next section.

Stream-Relational Algebra

Whereas the SQL-like syntax of the previous section is designed to be authored by humans, this section describes an algebra designed to be optimized and executed by stream-query engines. The algebra is stream-relational because it augments with stream operators the relational algebra from databases. Relational algebra has well-understood semantics (Garcia-Molina et al. 2008), and the CQL authors rigorously defined the formal semantics for the additional streaming operators (Arasu and Widom 2004). The following paragraphs provide only an informal overview of the operators; interested readers can consult the literature for formal treatments. The notation for operator signatures is:

$$\textit{operator}_{\langle \text{configuration} \rangle}(\textit{input}) \rightarrow \textit{output}$$

The configuration of an operator specializes its behavior. The input to an operator consists of one or multiple relations or a stream. And the output of an operator consists of either a relation or a stream. Relational algebra is compositional, since the output of one operator can be plugged into the input of another operator, modulo compatible kind and type. For instance, the stream-relational algebra for stream *Bargains* in the CQL example from the start of section "SQL-Like Syntax" is:

$$\textbf{istream}\left(\sigma_{\langle ask \leq low \rangle}\left(\bowtie_{\langle Quotes.ticker = History.ticker \rangle}\ \left(\textbf{now}(Quotes), History\right)\right)\right)$$

Classic relational algebra has operators from relations to relations (Garcia-Molina et al. 2008):

- $\pi_{\langle assignments \rangle}(relation) \rightarrow relation$
  Project each input tuple using assignments to create a tuple in the output relation.
- $\sigma_{\langle predicate \rangle}(relation) \rightarrow relation$
  Select tuples for which the predicate is true, and filter out tuples for which it is false.
- $\bowtie_{\langle predicate \rangle}(relation^{+,}) \rightarrow relation$
  Join tuples from input relations as if with a cross product followed by $\sigma_{\langle predicate \rangle}$.
- $\gamma_{\langle groupBy,assignments \rangle}(relation) \rightarrow relation$
  Group tuples and then aggregate within each group, using the given assignments.

For brevity, we omitted other classic relational operators, but they could be added trivially, thanks to the compositionality of the algebra. CQL can use the well-defined semantics of classic relational algebra operators by applying them on snapshots of relations at a point in time. Some operators, such as π and σ, process each tuple in isolation, without carrying any state from one tuple to another (Xu et al. 2013). These operators could be easily lifted to work on streams, and indeed, some streaming SQL dialects do just that. But the same is not true for stateful operators such as $\bowtie$ and γ. To use these on streams, CQL first converts streams to relations, using window operators:

- **now**$(stream) \rightarrow relation$
  At each time instant t, all tuples from the input stream with timestamp exactly t.
- **unbounded**$(stream) \rightarrow relation$
  At each time instant t, all tuples from the input stream with timestamp at most t.
- **range**$_{\langle partitionBy,size(,slide)^? \rangle}(stream) \rightarrow relation$
  Use a time-based sliding window on the input stream as the output relation.
- **rows**$_{\langle partitionBy,size(,slide)^? \rangle}(stream) \rightarrow relation$
  Use a count-based sliding window on the input stream as the output relation.

These window operators correspond directly to the corresponding surface syntax discussed in section "SQL-Like Syntax". Gedik surveyed these and more window constructs and their implementation (Gedik 2013). A final set of operators turns relations back into streams:

- **istream**$(relation) \rightarrow stream$
  Watch input relation for insertions, and send those as tuples on the output stream.
- **dstream**$(relation) \rightarrow stream$
  Watch input relation for deletions, and send those as tuples on the output stream.
- **rstream**$(relation) \rightarrow stream$
  At each time instant, send all tuples currently in input relation on output stream.

This article illustrated continuous queries in SQL-like languages using CQL as the concrete example, because it is clean and well studied. The original CQL work contains more details and a denotational semantics (Arasu et al. 2006; Arasu and Widom 2004). Soulé et al. furnish CQL with a static type system and formalize translations from CQL via stream-relational algebra to a calculus with an operational semantics (Soulé et al. 2016).

Findings

CQL was not the first dialect of SQL for streaming. TelegraphCQ reused the front-end of PostgreSQL as the basis for its surface language (Chandrasekaran et al. 2003). Rather than focusing on surface language innovation, TelegraphCQ focused on a stream-relational algebra back-end that pioneered new techniques for dynamic optimization and query sharing. Gigascope had its own dialect of SQL called GSQL (Cranor et al. 2003). Unlike CQL, GSQL used an algebra where all operators work directly on streams. As discussed earlier, this is straightforward for π and σ, but not for $\bowtie$ and γ. Therefore, GSQL required $\bowtie$ and γ to be configured with constraints over ordering attributes that effectively function as windows. Aurora used a graphical interface for

surface-level programming, but we still consider it an SQL-like system, because it used a stream-relational algebra (Abadi et al. 2003). Aurora's Stream Query Algebra (SQuAl) contained the usual operators π, σ, $\bowtie$, and γ, as well as union, sort, and a resample operator that interpolates missing values.

CQL took a more language-centric approach than its predecessors. It also inspired work probing semantic subtleties in SQL-like streaming languages. Jain et al. precisely define the semantics for the corner case of StreamSQL behavior where multiple tuples have the same timestamp (Jain et al. 2008). In that case, there is no inherent order among these tuples, so tuple-based windows must choose arbitrarily, leading to undefined results. Furthermore, if actions are triggered on a per-tuple basis, there can be multiple actions at a single timestamp, leading to spurious intermediate results that some would consider a glitch. The SECRET paper is also concerned with problems of time-based sliding windows (Botan et al. 2010). SECRET stands for ScopE (which timestamps belong to a window), Contents (which tuples belong to a window), REport (when to output results), and Tick (when to trigger computation). Finally, Zou et al. explored turning repeated SQL queries into continuous CQL queries by turning parameters that change between successive invocations into an input stream (Zou et al. 2010).

Today, there is still much active research on big-data streaming systems, but the focus has shifted from SQL dialects to embedded domain-specific languages (EDSLs, Hudak 1998). An EDSL for streaming is a library in a host language that offers abstractions for continuous queries. In practice, most EDSLs lack the rigorous semantics of stand-alone languages such as CQL but have the advantage of posing a lower barrier to entry for developers who are already proficient in the host language, being easier to extend with user-defined operators, and not requiring a separate language tool-chain (compiler, debugger, integrated development environment, etc.).

Examples

The beginnings of sections "SQL-Like Syntax" and "Stream-Relational Algebra" show a concrete example of the same query in CQL and in stream-relational algebra, respectively. The most famous example of a set of continuous queries written in an SQL-like language is the Linear Road benchmark. The benchmark consists of computing variable-rate tolling for congestion pricing on highways. A simplified version of Linear Road serves to motivate and illustrate CQL (Arasu et al. 2006). The full version of Linear Road is presented in a paper of its own (Arasu et al. 2004). Both the simplified version and the full version of the benchmark continue to be popular, and not just for SQL-inspired streaming systems and languages (Grover et al. 2016; Hirzel et al. 2016; Jain et al. 2006; Soulé et al. 2016).

Future Directions for Research

Stream processing is an active area of research, and new papers often use a streaming SQL foundation to present their results. One challenging issue for streaming systems is how to handle out-of-order data. For instance, CEDR suggests a solution based on stream-relational algebra using several timestamps per tuple (Barga et al. 2007). One challenge that is particular to SQL-like languages is how to extend them with user-defined operators without losing the well-definedness of the restricted algebra. For instance, StreamInsight addresses this issue with an extensibility framework (Ali et al. 2011). Finally, the semantics of SQL are defined by reevaluating relational operators on windows whenever the window contents change. Nobody suggests that this reevaluation is the most efficient approach, but developing better solutions is an interesting research challenge. For instance, the DABA algorithm performs associative sliding-window aggregation on FIFO windows in worst-case constant time (Tangwongsan et al. 2017). All three of the abovementioned research topics (out-of-order processing, extensibility, and incremental streaming algorithms) are still ripe with open issues.

Cross-References

- Big SQL
- Stream Processing Languages and Abstractions

References

Abadi DJ, Carney D, Cetintemel U, Cherniack M, Convey C, Lee S, Stonebraker M, Tatbul N, Zdonik S (2003) Aurora: a new model and architecture for data stream management. J Very Large Data Bases (VLDB J) 12(2):120–139

Ali M, Chandramouli B, Goldstein J, Schindlauer R (2011) The extensibility framework in Microsoft StreamInsight. In: International conference on data engineering (ICDE), pp 1242–1253

Arasu A, Widom J (2004) A denotational semantics for continuous queries over streams and relations. SIGMOD Rec 33(3):6

Arasu A, Cherniack M, Galvez E, Maier D, Maskey AS, Ryvkina E, Stonebraker M, Tibbetts R (2004) Linear road: a stream data management benchmark. In: Conference on very large data bases (VLDB), pp 480–491

Arasu A, Babu S, Widom J (2006) The CQL continuous query language: semantic foundations and query execution. J Very Large Data Bases (VLDB J) 15(2): 121–142

Barga RS, Goldstein J, Ali M, Hong M (2007) Consistent streaming through time: a vision for event stream processing. In: Conference on innovative data systems research (CIDR), pp 363–373

Botan I, Derakhshan R, Dindar N, Haas L, Miller RJ, Tatbul N (2010) SECRET: a model for analysis of the execution semantics of stream processing systems. In: Conference on very large data bases (VLDB), pp 232–243

Chandrasekaran S, Cooper O, Deshpande A, Franklin MJ, Hellerstein JM, Hong W, Krishnamurthy S, Madden S, Raman V, Reiss F, Shah MA (2003) TelegraphCQ: continuous dataflow processing for an uncertain world. In: Conference on innovative data systems research (CIDR)

Cranor C, Johnson T, Spataschek O, Shkapenyuk V (2003) Gigascope: a stream database for network applications. In: International conference on management of data (SIGMOD) industrial track, pp 647–651

Garcia-Molina H, Ullman JD, Widom J (2008) Database systems: the complete book, 2nd edn. Pearson/Prentice Hall, London, UK

Gedik B (2013) Generic windowing support for extensible stream processing systems. Softw Pract Exp (SP&E) 44:1105–1128

Grover M, Rea R, Spicer M (2016) Walmart & IBM revisit the linear road benchmark. https://www.slideshare.net/RedisLabs/walmart-ibm-revisit-the-linear-road-benchmark (Retrieved Feb 2018)

Hirzel M, Rabbah R, Suter P, Tardieu O, Vaziri M (2016) Spreadsheets for stream processing with unbounded windows and partitions. In: Conference on distributed event-based systems (DEBS), pp 49–60

Hudak P (1998) Modular domain specific languages and tools. In: International conference on software reuse (ICSR), pp 134–142

Jain N, Amini L, Andrade H, King R, Park Y, Selo P, Venkatramani C (2006) Design, implementation, and evaluation of the linear road benchmark on the stream processing core. In: International conference on management of data (SIGMOD), pp 431–442

Jain N, Mishra S, Srinivasan A, Gehrke J, Widom J, Balakrishnan H, Cetintemel U, Cherniack M, Tibbets R, Zdonik S (2008) Towards a streaming SQL standard. In: Conference on very large data bases (VLDB), pp 1379–1390

Soulé R, Hirzel M, Gedik B, Grimm R (2016) River: an intermediate language for stream processing. Softw Pract Exp (SP&E) 46(7):891–929

Tangwongsan K, Hirzel M, Schneider S (2017) Low-latency sliding-window aggregation in worst-case constant time. In: Conference on distributed event-based systems (DEBS), pp 66–77

Xu Z, Hirzel M, Rothermel G, Wu KL (2013) Testing properties of dataflow program operators. In: Conference on automated software engineering (ASE), pp 103–113

Zou Q, Wang H, Soulé R, Hirzel M, Andrade H, Gedik B, Wu KL (2010) From a stream of relational queries to distributed stream processing. In: Conference on very large data bases (VLDB) industrial track, pp 1394–1405