Journal of Universal Computer Science, vol. 12, no. 9 (2006), 1165-1176
submitted: 31/12/05, accepted: 12/5/06, appeared: 28/9/06 © J.UCS

Extension of CQL over Dynamic Databases

Antal Buza
(College of Dunaújváros, Hungary
buza@mail.duf.hu)

Abstract: CQL, Continuous Query Language is suitable for data stream queries. Sometimes it is better if the queries operate on relational databases *and* data streams simultaneously. The execution of a CQL query takes a long time (several hours, days or even more). It is not clear what kind of semantics is suitable for the user when the database is updated during the execution of a CQL query. In this paper we give a short description of CQL, a characterization of update-problems, and we offer possible suggestions for the semantic extension of CQL. After the expansion, the CQL would be suitable for solving much more practical problems. The parallel usage of continuous data streams and updatable databases would be settled.

Keywords: data-flow languages, database, query languages
Categories: D.3.2, H.2.1, H.2.3

1 Introduction

CQL, Continuous Query Language is an expressive SQL-based declarative language for registering continuous queries against streams and updatable relations. CQL is suitable for data stream queries. Sometimes it is better if the queries operate on relational databases *and* data streams simultaneously. The execution of a CQL query takes a long time (several hours, days or even more). It is not clear what kind of semantics is suitable for the user when the database is updated during the execution of a CQL query. For example, if correctly applied, changing the value of an account while the official rates are updated, then the CQL system calculates with the retroactive effect of the update. For observing the trade of a supermarket, another semantics is required when the prices are changed while using the CQL query. In this case, the effect of the update is valid from the moment of the update. In this paper we give a short description of CQL, a characterization of the update-problems, and we offer possible suggestions for the semantic extension of CQL.

Another interesting problem is the explanation of the consistent state. In classical database theory, it is a usual requirement that the 'normal' state of a database is the consistent state. However, the consistency is not a permanent state, during the updates the consistent state may be briefly damaged. We have investigated what the effect of the inconsistent state is on the currently operating CQL queries.

In [Section 2], mainly based on [Arasu, 03c] and [Babu, 01b], we give a short overview of data streams and the Continuous Query Language. It seems the developing process of CQL has not yet finished; as a number of questions have not been solved yet or the research and development group analyses of different situations, solutions, and the effects of these, has not been completed. In [Section 3] some extensions to CQL are suggested. The extensions follow the effects of updates of databases realized during the long execution of CQL queries. In real situations, the effects of the updates of databases have three forms: the 'retroactive', the 'from the update', and the 'strong' effect of the update.

2 Introduction to continuous queries and CQL

In this section we introduce the readers to data streams, continuous queries and the Continuous Query Language (CQL). This section is based on technical literature and we recommend that it is read by those who are not familiar with CQL.

2.1 The data stream and the continuous query

Data stream: Several applications naturally generate data streams as opposed to data sets: financial tickers, performance measurements in network monitoring and traffic management, log records, manufacturing processes, data feeds from sensor applications, email messages and others. We can claim that the data streams are more natural than databases.

Continuous query: We consider the *continuous query* which is issued once and then logically run continuously over the data stream and/or over the database (in contrast to traditional *one-time* queries which are run once to completion over the current data sets).

We can consider/transform the database into the stream (for example as a result of some kind of sequential read of the database), and conversely, using the data stream records we can append the database.

Why do we use data streams and continuous queries? Why it is not a good strategy to build databases and use classical SQL queries? Because it may happen that the data stream produces a very huge mass of data in a short time. For example: we would like to detect the daily traffic on a 2Gbit/s line; which might generate more than 100Mbyte/s of data, i.e. 6Gbyte/min, 360Gbyte/hour, 8.6Tbyte/day. The store (the speed of update and the disk capacity) and the repeated (!) query of this huge mass of data causes several problems, especially when we use the join operator. A better way would be to use a continuous query over the data stream.

2.2 CQL - Continuous Query Language

"CQL is an expressive SQL-based declarative language for registering continuous queries against streams and databases" – Jennifer Widom. CQL is developed and implemented at Stanford University in the STREAM project.

2.3 Illustrative CQL examples

Instead of the full description of CQL, we cite some illustrative examples to demonstrate the abilities of CQL. Consider the domain of network traffic management for a large network. The network traffic management applications processes are typically rapid, unpredictable and continuous data streams. In the following examples we observe the traffic generated streams PT_c and PT_b (packet traces collected from the costumer and backbone links, respectively). For simplicity, we assume that the packet header comprises the fields: *saddr* – IP address of packet sender, *daddr* – IP address of packet destination, *id* – packet identification number, *length* – length of the packet, *timestamp* – time when the packet header was recorded.

The first CQL example computes the load on the backbone link averaged over one minute periods and notifies the network operator if the load exceeds a threshold *T*.

```
SELECT notifyoperator(sum(length))
   FROM $PT_b$
   GROUP BY getminute(timestamp)
   HAVING sum(length)> $T$
```

In this example, the `notifyoperator` and the `getminute` are self-explanatory functions. Similar functionality might be achievable using triggers in conventional DBMS, but this is certainly something conventional triggers are not designed for.

The next example illustrates the finding of the fraction of traffic on the backbone link coming from the customer network.

```
(SELECT count(*) FROM $PT_c$ AS C, $PT_b$ AS B
     WHERE C.saddr=B.saddr AND
           C.daddr=B.daddr AND
           C.id=B.id)
/ (SELECT count(*) FROM $PT_b$)
```

This is an example of an ad-hoc continuous query. (Since unbounded intermediate storage could potentially be required for joining two continuous data streams, we must use some kind of restriction: such as a time window, because the answer might only be an approximate answer).

The third example monitors the top 5% source-to-destination pairs in terms of traffic on the backbone link:

```
WITH load AS
  (SELECT saddr,daddr,SUM(length) AS traffic
    FROM $PT_b$
    GROUP BY saddr,daddr)
  SELECT saddr,daddr,traffic
    FROM load AS $L_1$
    WHERE (SELECT COUNT(*)
           FROM load AS $L_2$
           WHERE $L_2$.traffic < $L_1$.traffic)>
                (SELECT 0.95*COUNT(*) FROM load)
    ORDER BY traffic
```

In the above example we used the SQL3 WITH construct for ease of expressing the query.

As stated in [Section 1], the continuous queries operate either on the relational databases or on the data streams separately or on both simultaneously.

3 The updates of the databases produce different effects on answer strategy of the CQL queries

CQL was developed mainly for data stream processing. However, tasks which require the common and simultaneous usage of the data streams and databases in practice may occur. The number of tasks keeps growing, because data stream generator equipments are being continuously developed; therefore they will be used in more and more fields. As a consequence, data stream processing will appear in more and more such applications; where the stored (e.g. 'historical') data produced by other systems are required to generate the correct answer. The 'historical' data is typically stored in databases and/or files.

Simultaneous usage of the data stream and the database raises the question: "How the CQL answer is affected by the updates of the database during the long time execution of the CQL query?" In this section, examples are used to demonstrate that the update of the database must been taken into account differently in each particular case in order to produce the correct answer for the customer. It is probable that this problem has not yet been studied because non-database oriented questions have only been considered in a basic manner. Therefore the effects of updates have not yet been analyzed.

The aim of the paper is to suggest the extension of CQL, because mixed (stream and database) applications would seem to be very important in the near future – as illustrated by these experiments.

Remember that the continuous query runs continuously for a long time. During this long running time the used database(s) might be updated. How must the CQL query for the event of the update reflect this?

3.1 The retroactive effect of the update

Let us take an example in the banking environment. We store the accounts in the RDBMS, in the relation "`ACC`", several updates come through the stream "`UPDS`" and we store the rates in the relation "`RAT`". The rates may be changed during the execution of the CQL query. In this case, when we use a continuous query for the actual sum of the accounts in € (or in $, or in any common value), we can calculate with the retroactive effect of the update of rates relation. For example, we have an account of $400, and supposing $1=€*N* before the change of the rate. However, the rate suddenly changes so that the new rate is $1=€*M*. As a consequence the value of our account expressed in € will change.

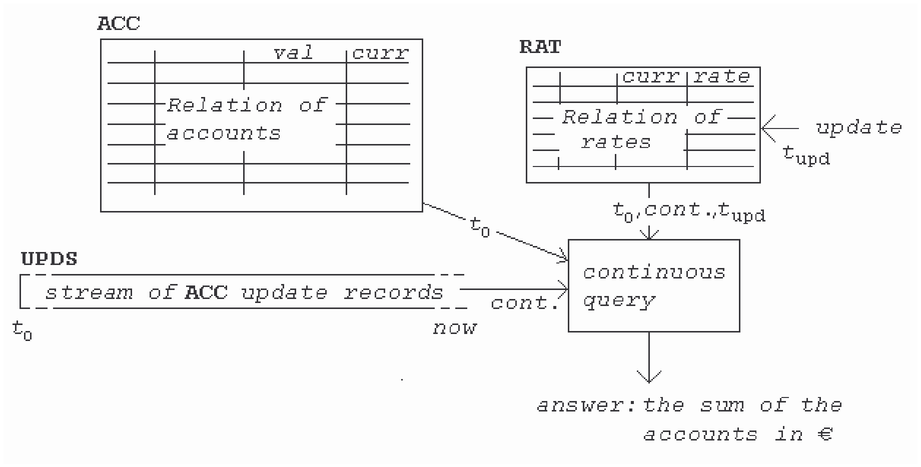

Figure 1: The effect of the update of `RAT` *relation is retroactive for the answer of the CQL query*

The situation is illustrated by [Figure 1]. The accounts are stored in the relation `ACC`, the rates are stored in the relation `RAT`, and the update records of the `ACC` relation come through the `UPDS` data stream. The continuous query was started at time t_0, and the relation `RAT` was updated at time t_{upd}.

The CQL query seems to be extendible in order to manage the above mentioned problem. The suggested extension of the CQL query is as follows: the query summarizes the actual accounts - stored in `ACC` relation - expressed in Euro. The query is sensitive to the update of the rates (`RAT`) relation.

```
SELECT SUM(val*rate)
   FROM (STREAM(ACC) CONTINUE UPDS)
        NATURAL JOIN
        RAT(RETROACTIVE)
```

Operator `STREAM` produces a stream from the relation `ACC`. The keyword `CONTINUE` symbolizes a 'time-ordered' `UNION` (e.g. one stream is described in the clausal `FROM`, first part of the stream consists of the rows of the `ACC` relation at time t_0, followed by the stream records coming through the `UPDS` data stream). Keyword `RETROACTIVE` indicates that the continuous query will be virtually re-started when the relation `RAT` is updated (The 'virtual re-start' means that the system re-reads all relations and one processes the stream from now). The virtually re-started state of the continuous query is illustrated by [Figure 2].

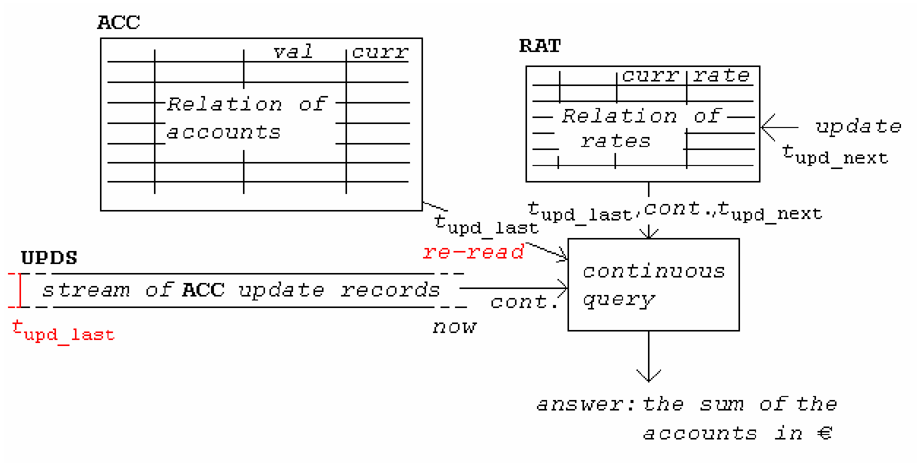

Figure 2: The virtually restart of the `UPD_RETRO` *type CQL query*

The t_{upd_last} symbolizes the last update-time of the `RAT` relation, the t_{upd_next} symbolizes the next update time of the `RAT` relation (in the future), thus t_{upd_last} < now < t_{upd_next}. The virtual re-start means that the query in the moment of time t_{upd_last} re-reads the relation ACC and it uses only the new/actual part of the `UPDS` stream (produced after moment of time t_{upd_last}). The query reads the `RAT` relation logically permanently. The theoretical operation of the continuous query is based on this fact. In practice, when there is sufficient memory for the storage of the relation `RAT;` then the query does not read this relation continuously or repeatedly, but does it immediately after the last update of the relation.

In the above example, the identified problem may be solved by repeating the 'classical' SQL query frequently or continuously. However, in cases where the table "`ACC`" is very large and/or is distributed over a number of geographical places, then the usage of the CQL query is much more suitable than using the SQL query. In this case, the using the CQL query requires less resources than the SQL query, therefore the CQL query generates the answer more quickly.

3.2 The 'from now' type effect of the database update

Another required semantics of the update is explained as follows: when the update produces an effect only from the actual time (from the moment of the update) and in the future of course. For example, in trade systems when the prices are changed and the continuous query is used for determining the actual (daily, weekly, monthly) income, we can calculate the effect of the update only from now (from the moment of the update) and in the future (till the next update).

Let us look at an example for cases like this. [Figure 3] shows the environment of the continuous query.

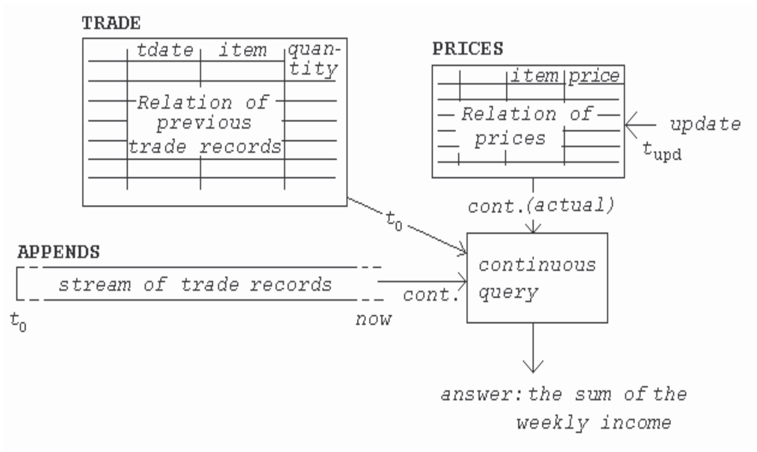

Figure 3: The environment of the continuous query when the effect of the update belongs to the type 'from now'

In [Figure 3] the relation `TRADE` is the relation of the previous trade records, in the relation `PRICES` we store the actual prices and the new trade records come continuously through the data stream `APPENDS`. We can calculate the sum of the weekly income using the following continuous query:

```
SELECT SUM(quantity*price)
  FROM (STREAM(SELECT * FROM TRADE
                WHERE WEEK(tdate)=WEEK(date))
        CONTINUE APPENDS)
       NATURAL JOIN
        PRICES(ACTUAL)
```

Now the keywords `STREAM` and `CONTINUE` are the same as in the previous example, the keyword `ACTUAL` indicates that the query uses the actual state of the relation (which is changeable during the execution of the continuous query).

This query calculates the weekly income of a supermarket, taking into account the changes of the prices of the sold items. The effects of the updates of the prices are valid from the time of the updates till the next updates.

The query execution system rereads the relation of prices repeatedly or it does so when the system detects the update of this relation, respectively, depending on the size of the updated relation and the size of the usable memory.

In this example, the common use of the "`TRADE`" database table and the stream records is reasonable because we can put a question referring to some earlier period than the last start of the CQL query. Parallel usage of table "`TRADE`" and the stream "`APPENDS`" is not really simultaneous mode. Usage of the table "`PRICES`" is trivial; indeed the really simultaneous mode is with the stream "`APPENDS`".

3.3 Strongly sensitive update queries

The third strategy is that when the query is strongly sensitive to the update.

Examples are found for such cases in trade systems: when the names of manufacturers have been changed or firms have been merged or separated and the new organisation will take over the continuity of the predecessors' debts. In this situation the continuous query, being focused on determining the actual partner's invoices, is strongly sensitive to the updates described above.

The query uses the original (i.e. the contents of the relation at the moment of the start of the CQL query) contents of the relations. The execution of the continuous query terminates when the content of those relations is changed. The query may be restarted manually or automatically; and it produces a new answer that is totally independent of the previous answer.

The following form is suggested to specify the relation update that is the cause of the strongly sensitive queries:

```
Relation(ORIGINAL)
```

The keyword `ORIGINAL` indicates that the continuous query uses the original content of the relation during the executions. The update of this relation must cause the termination of the execution of the continuous query. As we have already discussed; the query may be restarted automatically or manually.

The three semantics ('update retro', 'actual' and 'original') are not mutually exclusive; it means that all versions may be used in the same query too.

3.4 Qualification by attributes

From the point of view of the continuous query the wanted effects of the updates of the relations may be different according to the different attributes. A continuous query which is sensitive to the update of the price of any item, at the same time one is not sensitive to the update of the supplier's phone number. As a consequence, the wanted effects of the updates of the relations must be determined by the attributes. The possible general form of the qualification described in CQL is as follows:

```
Table_name( [(attribute list)RETROACTIVE,]
            [(attribute list)ACTUAL,]
            [(attribute list)ORIGINAL] )
```

The update of the non-referred attributes has no effects during the execution of the actual CQL query.

When the wanted effect of the update is independent of the specific attributes, the previously used shorter form is also suggested:

$$\text{Table\_name}\left(\begin{bmatrix}\text{RETROACTIVE}\\ \text{ACTUAL}\\ \text{ORIGINAL}\end{bmatrix}\right)$$

Using the attribute-depending form, the previous examples are transformed into the following forms:

```
SELECT SUM(val*rate)
    FROM (STREAM(ACC) CONTINUE UPDS)
         NATURAL JOIN
         RAT((rate)RETROACTIVE)
```

and

```
SELECT SUM(quantity*price)
  FROM (STREAM(SELECT * FROM TRADE
                WHERE WEEK(tdate)=WEEK(date))
        CONTINUE APPENDS)
       NATURAL JOIN
        PRICES((price)ACTUAL)
```

As an example for the mixed update-sensitivity, examine the following query:

```
SELECT SUM(quantity*price*rate)
  FROM ((STREAM(TRADE)
        CONTINUE APPENDS)
      NATURAL JOIN
        PRICES((price)ACTUAL))
      NATURAL JOIN
        RAT((rate)RETROACTIVE)
```

This continuous query calculates the value of the actual inventory of a supermarket expressed in Euro. (The prices and the rates might be changed while the value of the trade is expressed in a different currency.)

3.5 Syntax of the expansion of the CQL

The formal syntax of the suggested expansion of CQL is as follows: in every case when we refer to the relations – similarly to SQL – the usage of the following form is permitted:

$$\text{Table\_name}\left(\{(\text{attribute list})\}\begin{bmatrix}\text{RETROACTIVE}\\ \text{ACTUAL}\\ \text{ORIGINAL}\end{bmatrix}\right)$$

In this way we can instruct the CQL system what type of reaction is required when the CQL system detects the update of the specific (or any) attributes of specific relations.

3.6 Motivation of the introduction of the new keywords

Traditionally the database query operates on the actual state of the database and produces the answer as soon as possible. Therefore it is not necessary to initiate the qualification ‘actual’. In fact the qualification ‘actual’ might be the default qualification applicable to all other database-actions. The executions of the CQL queries have several differences compared to the executions of the SQL queries. One of the considerable differences is the execution-time of the query. The execution-time of the SQL query is short (prompt, as soon as possible), in contrast with the long execution-time of the CQL query. Therefore it would be reasonable for the usage of keywords for the time-qualification. The same introduction of the three new qualifications (‘retroactive’, ‘actual’, and ‘original’) is required. In this manner names have been given for all occurring cases.

The questions using the ‘actual’ qualifications can also be given in other ‘traditional’ forms too. There are other examples for the possibility of different formulations of the same query in the database environment. For example: the ‘cross join’ (and other types of join operators) in SQL2 are definitely not necessary; because there are another equivalent formulae. Nevertheless it is true, that the ‘join’ operators are very usable. In time-dependent applications, usages of all three qualifications are suggested for the differentiation of the causes.

3.7 The expressive power and the practicability of the expanded CQL

The suggested expansion of CQL allows of the usage of the new type queries (with new semantic denotation). As a consequence, the expansion causes the increasing of the theoretical expressive power partway. On the other hand the expansion increases the practical usability of CQL too.

3.8 The accuracy of the answer and the inconsistency state of the database

There are several reasons which cause inaccuracies in the answer. These inaccuracies do not necessary decrease the usefulness of the answer. In most cases, the approximate answer may also be suitable too. For example: it is not required to process all items when the focus is only on the trend (traffic or market for instance). Of much more importance is the recognition of the main characteristics of the processes. It may happen that we lose a part of the data stream. As we can or cannot store and use the records from the data stream from the termination-time of the query execution till the restart of the query execution This is just one of several reasons why the answer should be only approximated – but nevertheless it is suitable.

Another reason for inaccuracy is that the continuous queries work is bounded by memory and time. The bounds are not only technical, but also theoretical, too. When working with a big time-window, the recognition efficiency of the essential processes may be damaged. The most effective time-window size strongly depends on the real problem, and one must be based on the theoretical considerations rather than the technical conditions.

During the execution of the continuous query there must be an operational pause to enable:

- the time-requirement of the virtual restart in the case of `RETROACTIVE`,
- the time-requirement of the reread in the case of `ACTUAL`,
- the time-requirement of the restart in the case of `ORIGINAL`.

During this pause, a partial loss of the stream may occur, especially when the data stream is coming very fast. In cases where the lost part of the data stream would cause any essential problems; these must be avoided by saving and processing of that part of the stream.

Another interesting area is the area of inconsistency. It is a usual requirement in classical database theory that the 'normal' state of a database is the consistent state. The consistency is not a permanent state. During updates it may become damaged for a brief period of time. The inaccuracy of the CQL queries currently operating is the side effect of the inconsistent state of the database.

The undoing of updates to the databases – especially by the cascaded revoke – might cause a decrease in the accuracy of the database. During the execution of the continuous query these situations might perhaps be manageable, but when the continuous query execution is stopped exactly before a cascaded revoke, then the accuracy of the answer would be reduced. Probably, general solutions for avoiding inaccurate results might not be given. At least, giving a signal to the user is required when any incident, that has an influence over the accuracy of the answer, has occurred during the execution of the continuous query.

4 Conclusion

The continuous data streams are native data occurrences. Perhaps in lot of cases, they are much more native than the relations. Thus the usage of data streams has a number of advantages. Therefore, the expansion of CQL - since it is suitable for the management of real situations - might not only be useful but also very desirable. It is in this light that this paper makes suggestions for the expansion of CQL.

References

[Arasu, 03a] Arvind Arasu and Shivnath Babu and Jennifer Widom: An abstract Semantics and Concrete Language for Continuous Queries over Streams and Relations. Invited paper in the DBPL (9th International Conference on Data Base Programming Languages) workshop, September 2003. 12p. (http://dbpubs.stanford.edu/pub/2002-57)

[Arasu, 03b] Arvind Arasu et al.: STREAM: The Stanford Stream Data Manager. IEEE Data Engineering Bulletin, Vol. 26 No. 1, March 2003. 8p. (http://dbpubs.stanford.edu/pub/2003-21)

[Arasu, 03c] Arvind Arasu and Shivnath Babu and Jennifer Widom: The CQL Continuous Query Languag: Semantic Fundations and Query Execution. Proceedings of the 9th International Conference on Data Base Programming Languages (DBPL) September 2003. 32p. (http://dbpubs.stanford.edu/pub/2003-67)

[Babu, 01a] Shivnath Babu, Lakshminarayanan Subramanian and Jennifer Widom: A Data Stream Management System for Network Traffic Management. In Proc. of the Workshop on Network-Related Data Management (NRDM 2001), May 2001. 2p. (http://dbpubs.stanford.edu/pub/2001-20)

[Babu, 01b] Shivnath Babu and Jennifer Widom: Continuous Queries over Data Streams. SIGMOD Record, Sept. 2001. 17p. (http://dbpubs.stanford.edu/pub/2001-9)

[Motwani, 03] Rajeev Motwani et al.: Query Processing, Resource Management, and Approximation in a Data Stream Management System. In Proc. of the 2003 Conf. on Innovative Data Systems Research (CIDR), January 2003. 12p.
(http://www-db.cs.wisc.edu/cidr/cidr2003/program/p22.pdf)

[Srivastava, 03] Utkarsh Srivastava, Shivnath Babu and Jennifer Widom: Monitoring Stream Properties for Continuous Query Processing. In Proc. of the 2003 Workshop on Management and Processing of Data Streams (MPDS 2003), June 2003. 5p.
(http://dbpubs.stanford.edu/pub/2003-23)