A Theory of Stream Queries

Yuri Gurevich[1], Dirk Leinders[2], and Jan Van den Bussche[2]

[1] Microsoft Research
gurevich@microsoft.com
[2] Hasselt University and Transnational University of Limburg
{dirk.leinders,jan.vandenbussche}@uhasselt.be

Abstract. Data streams are modeled as infinite or finite sequences of data elements coming from an arbitrary but fixed universe. The universe can have various built-in functions and predicates. Stream queries are modeled as functions from streams to streams. Both timed and untimed settings are considered. Issues investigated include abstract definitions of computability of stream queries; the connection between abstract computability, continuity, monotonicity, and non-blocking operators; and bounded memory computability of stream queries using abstract state machines (ASMs).

1 Introduction

Over the past few years in the database systems research community, much attention has been paid to query languages and query processing for data streams. We give just a few references here [15,5,6,14,7]; much more has been published. Stream queries are typically "continuous" in that their result must be continually updated as new data arrives: indeed, stream applications are "data-driven". Consequently, continuous stream queries must be computed in an incremental fashion, using so-called "non-blocking" operators. Relational algebra operators that are monotone are non-blocking; query operators that are not monotone, such as difference, or grouping and aggregation, are typically made non-blocking by restricting them to sliding windows.

The aim of this paper is to offer a theoretical framework that attempts to clarify various philosophical questions about stream queries. For instance, if streams are thought of as infinite, and arbitrary queries are modeled as functions from streams to streams, what does it mean for a query to be computable? Is computability the same concept as continuity? What is the precise connection between continuity and monotonicity? Can one give a formal definition of what it means for an arbitrary operator to be non-blocking?

Earlier work in this direction has already been reported by Arasu and Widom [3] and by Law, Wang and Zaniolo [12]. Our work has the following new features:

1. We distinguish from the outset between timed and untimed applications. In a timed setting, the timestamps in the output stream of some stream query are synchronized with the timestamps in the input stream; in an untimed setting, they are not. The usual applications mentioned in the data stream literature,

M. Arenas and M.I. Schwartzbach (Eds.): DBPL 2007, LNCS 4797, pp. 153–168, 2007.

such as stock quotes or sensors, are timed. Nevertheless, untimed streams also find applications, e.g., in audio or video streams, or Internet broadcasts, where the logical order among arriving packets is more important than precise timing information. More fundamentally, however, much of the theory of stream queries can already be developed on the more basic untimed level, viewing timed streams merely as a special case of untimed streams. Nonetheless, we will also identify some specific aspects of timed queries, in particular, their non-predicting nature (in a sense that will be made precise later).

2. Our formal definitions of abstract computable stream queries are grounded in the theory of type-2 effectivity (TTE) [16]. This is a well-established theory of computability on infinite strings (and much more, which we will not use here). The basic idea of TTE, strikingly analogous to the idea of continuous stream queries, is that arbitrary long finite prefixes of the infinite output can be computed from longer and longer finite prefixes of the infinite input. A basic insight from TTE is that computable functions on infinite strings are indeed "continuous", but now in the precise sense of mathematical topology. More specifically, under a natural metric on infinite strings (known as the Cantor metric), where two strings are closer the longer they agree on their prefixes, computable functions can be shown to be continuous in the standard mathematical sense of the word. Continuity is a useful property for it provides us with a principled way to prove that not just *any* function from streams to streams can be naturally considered to be a stream query.
3. Our theory is abstract in the sense that elements from a stream can come from an arbitrary universe, equipped with predicates and functions. In mathematical logic one speaks of a *structure*, and we will refer to the universe as the *background structure*. In particular, we do not concern ourselves with the encoding of stream elements as bitstrings (finite or infinite), or with Turing machine computations on those bitstrings, since those aspects are already well understood from the TTE. Consequently, our theory is very general, and computable stream queries will turn out to be the same thing as continuous functions from streams to streams (where we introduce a variant of the Cantor topology that accommodates finite as well as infinite streams).
4. We define a concrete computation model for stream queries, called "streaming ASM". Due to the abstract nature of our theory, streaming ASMs are naturally based on (sequential) Abstract State Machines [9,10]. Every computable stream query is computable by a streaming ASM with an appropriate background structure. Moreover, streaming ASMs allow us to prove impossibility results. Specifically, we focus on bounded memory machines: such machines can only remember a constant number of previously seen stream elements. Bounded memory machines are natural in the context of query processing; for example, any query operator that applies a sliding window (typical in streaming applications) is computable in bounded memory. Bounded memory evaluation of stream queries was already emphasized by Arasu et al. [1]. We will prove that there exist simple queries that are not bounded memory computable, one of the simplest being the query INTERSECT: finding the common elements in two interleaved streams.

The present paper is a companion to our paper with Grohe, Schweikardt, and Tyszkiewicz [8] on Finite Cursor Machines (FCM). There, we studied classical database query processing using a bounded number of one-way cursors over the database relations. Streaming ASMs are similar to FCMs, but a crucial difference is that a streaming ASM has only one cursor, and this cursor is manipulated by the stream rather than by the machine (recall the data-driven nature of streaming applications). So, FCMs are unrealistically powerful in the context of data streams, because they can control their own cursors, which is only realistic when the stream is fully given as a completed, finite input list. In particular, FCMs are certainly at least as powerful as streaming ASMs. Since we already know that the query INTERSECT mentioned above is not even computable by an FCM, it follows by a reduction that the query is also not computable by a streaming ASM. Yet, in this paper we will give a direct proof of this result, that is much simpler and thus provides more direct insight on the limitations of bounded memory stream processing. Moreover, we will see that there exist stream queries that are computable by an FCM, but not by a streaming ASM.

We must also note that Arasu et al. have already presented impossibility results for bounded memory evaluation of stream queries [1], which seem to encompass, for example, the result on the query INTERSECT which we prove in this paper. Their impossibility proofs, however, assume that stream elements are encoded in bits: they show, for instance, that INTERSECT cannot be computed in $o(n)$ bits of memory. Our proof shows how to perform such impossibility arguments on a level where elements can be stored in their entirety as abstract objects. Note that it is not so easy to reduce the abstract level to the bit level, because in $o(n)$ bits of memory we cannot simply encode on the fly all elements we encounter as binary numbers, and still remember whether we have already seen some element earlier. More generally, there seems to be a discrepancy in the mentioned paper [1] between the computation model assumed for the positive results, and that assumed for the negative results (that for the negative results appears weaker).

2 Abstract Computability

Basically, we assume a universe $\mathbb{U}$ of data elements. A *stream* is a possibly infinite sequence of data elements. The set of all streams is denoted by *Stream*, and the set of all finite streams is denoted by *finStream*. Thus *finStream* $\subseteq$ *Stream*. We denote the i-th element of a stream $\mathbf{s}$ by s_i.

Our model of streams is very abstract and thus very general.

Example 1. Consider measurements coming from sensors, where each entry is a pair of the form (i, m) with i a sensor identifier and m a measurement. Suppose, at each discrete time point t (with time points modeled by natural numbers), we collect all entries that arrived in the interval $(t-1, t]$. Then $\mathbb{U}$ would contain sets of entries as data elements.

In a setting where time points would be more fine-grained, so that at most one entry can arrive per clock tick, $\mathbb{U}$ would contain entries directly as data elements, plus possibly some dummy element to indicate no entry arrived. □

Mathematically, a *stream query* is simply a mapping from *Stream* to *Stream*. Not all such mappings make sense in the streaming context, however. To make formal which queries do make sense, we define the notion of *abstract computability*. Intuitively, a stream query $\mathcal{Q}$ is abstract computable if there exists a function K: *finStream* → *finStream* such that the result of $\mathcal{Q}$ can be obtained by concatenating the results of K applied to larger and larger prefixes of the input.

Formally, for any K as above, we define the function

$$\text{Repeat}(K)\colon \mathbf{s} \mapsto \bigodot_{k=0}^{\text{size}(\mathbf{s})} K(\mathbf{s}^{\leq k}) \quad \text{of type} \quad \mathit{Stream} \to \mathit{Stream},$$

where $\mathbf{s}^{\leq k}$ is the prefix of $\mathbf{s}$ of length k, and size($\mathbf{s}$) is the length of $\mathbf{s}$ in case $\mathbf{s}$ is finite, and ∞ in case $\mathbf{s}$ is infinite (in which case the index k ranges over all natural numbers). Here $\bigodot$ denotes concatenation. We now define:

Definition 2. *A query $\mathcal{Q}$: Stream → Stream is abstract computable if there exists a function K such that $\mathcal{Q} = \text{Repeat}(K)$. We call K a kernel for $\mathcal{Q}$.*

The following example shows an abstract computable stream query:

Example 3. Let $\mathcal{Q}$ be the *running average* query, defined on streams of natural numbers and returning at each step the average value of the numbers arrived so far. The function returning $(\sum u_i)/n$ on input stream $u_1 \ldots u_n$ (and returning the empty stream when the input is the empty stream) is a kernel for $\mathcal{Q}$. □

In connection to finite streams, we make the following two important observations:

1. The answer to an abstract computable query on an infinite stream can be finite.

   Example 4. Consider the query $\mathcal{Q}$ that returns all elements in the input stream satisfying a certain predicate P. On a stream with only a finite number of elements satisfying P, the result of $\mathcal{Q}$ will be finite. Note that this query $\mathcal{Q}$ indeed has a kernel: for example the function K that given a finite stream, returns its last element if it satisfies P, and returns the empty stream otherwise, is a kernel for $\mathcal{Q}$. □

2. *The answer to an abstract computable query on a finite stream must be finite.* Indeed, the result of K is always a finite stream and on a finite input stream, K is applied only a finite number of times. So, queries transforming finite streams into infinite streams will never be computable in our model. This is not a problem since our model is primarily meant to capture input-data-driven computations.

We note:

Proposition 5. *Abstract computable stream queries are closed under composition.*

3 Continuity

We will now see that abstract computability and continuity of stream queries coincide.

Recall from elementary calculus [4] that a real function $f\colon \mathbb{R} \to \mathbb{R}$ is called continuous if for all $x \in \mathbb{R}$, for every neighborhood around $f(x)$, there exists a neighborhood around x that is completely mapped into the neighborhood of $f(x)$. In order to generalize this definition of continuity to stream queries, we must first agree on a definition of neighborhood of a stream $\mathbf{s}$. In other words, we need to define a suitable topology on streams.

For infinite streams, there is a standard topology, known from computable analysis [16], called the Cantor topology. This topology arises from the following metric (distance function) on infinite streams:

$$d(\mathbf{s}, \mathbf{s}') = \begin{cases} 0 & \text{if } \mathbf{s} = \mathbf{s}', \\ 2^{-n} & \text{if } \mathbf{s} \neq \mathbf{s}' \text{ and } n = \min\{i \mid s_i \neq s'_i\}. \end{cases}$$

According to this topology, open balls around an infinite stream $\mathbf{s}$ are sets of the form $\mathbf{B}(\mathbf{p})$, with $\mathbf{p}$ some finite prefix of $\mathbf{s}$, defined as follows:

$$\mathbf{B}(\mathbf{p}) = \{\mathbf{s}' \text{ infinite stream} \mid \mathbf{p} \text{ is a prefix of } \mathbf{s}'\}.$$

In this paper, we generalize this notion of open ball to the setting of both finite and infinite streams, as follows:

Definition 6. *Let* $\mathbf{p} \in$ *finStream. Then*

$$\mathbf{B}(\mathbf{p}) := \{\mathbf{s}' \in \mathit{Stream} \mid \mathbf{p} \textit{ is a prefix of } \mathbf{s}'\}.$$

Any set of the form $\mathbf{B}(\mathbf{p})$*, for some* $\mathbf{p} \in$ *finStream, is called an open ball. Elements of* $\mathbf{B}(\mathbf{p})$ *are called continuations of* $\mathbf{p}$*.*

This notion of open ball gives rise to a topology on streams, and the notion of continuity then amounts to the following:

Definition 7. $\mathcal{Q}\colon \mathit{Stream} \to \mathit{Stream}$ *is continuous if for every open ball* $\mathbf{B}$*, the pre-image* $\mathcal{Q}^{-1}(\mathbf{B})$ *is a union (possibly infinite) of open balls.*

Remark 8. The Cantor metric described above has only been defined on infinite streams. One may wonder whether the topology on *Stream* given by Definition 6 can be given by some metric of that sort but applicable to finite as well as infinite streams. The answer is negative: metrizable topologies must be Hausdorff, and our topology is not. Indeed, an infinite stream $\mathbf{q}$ and a finite prefix $\mathbf{p}$ of $\mathbf{q}$ can not be separated as each open ball containing $\mathbf{p}$ contains $\mathbf{q}$. For basic background on topology, we refer to Hocking and Young [11].

Theorem 9. *Let* $\mathcal{Q}$ *be a stream query mapping finite inputs to finite outputs. Then* $\mathcal{Q}$ *is abstract computable if and only if* $\mathcal{Q}$ *is continuous.*

Proof. For the only-if direction let K be a kernel for $\mathcal{Q}$, i.e., $\mathcal{Q} = \text{Repeat}(K)$. Consider $\mathbf{X} := \mathbf{B}(\mathbf{p})$. Let $\mathbf{s}$ be a stream in $\mathcal{Q}^{-1}(\mathbf{X})$. Then, from some natural number ℓ on, we know that $\bigodot_{k=0}^{\ell} K(\mathbf{s}^{\leq k})$ starts with $\mathbf{p}$. Consider then the open ball $\mathbf{B}(s_1 \dots s_\ell)$. Every $\mathbf{s}' \in \mathbf{B}(s_1 \dots s_\ell)$ is mapped into $\mathbf{X}$. Indeed,

$$\mathcal{Q}(\mathbf{s}') = \bigodot_{k=0}^{\ell} K(\mathbf{s}^{\leq k}) \odot \bigodot_{k=\ell+1}^{\text{size}(\mathbf{s})} K(\mathbf{s}'^{\leq k})$$

clearly starts with $\mathbf{p}$. Thus, $\mathbf{s} \in \mathbf{B}(s_1 \dots s_\ell) \subseteq \mathcal{Q}^{-1}(\mathbf{X})$, as desired.

For the if-direction, we define a kernel K for Q as follows. $K(()) := \mathcal{Q}(())$, and $K(\mathbf{s}u) := \mathcal{Q}(\mathbf{s}u) - \mathcal{Q}(\mathbf{s})$, where the difference is to be interpreted as removing a prefix, so that $\mathcal{Q}(\mathbf{s}u) = \mathcal{Q}(\mathbf{s}) \odot K(\mathbf{s}u)$. Note that $\mathcal{Q}(\mathbf{s})$ and $\mathcal{Q}(\mathbf{s}u)$ are both finite.

For K to be well-defined, we must show that $\mathcal{Q}(\mathbf{s})$ is indeed a prefix of $\mathcal{Q}(\mathbf{s}u)$. Consider $\mathbf{X} = \mathcal{Q}^{-1}(\mathbf{B}(\mathcal{Q}(\mathbf{s})))$. By continuity, $\mathbf{X}$ is a union of open balls. Thus, there must be an open ball $\mathbf{B}(\mathbf{p})$ with $\mathbf{s} \in \mathbf{B}(\mathbf{p}) \subseteq \mathbf{X}$. Clearly, $\mathbf{p}$ must be a prefix of $\mathbf{s}$. But then also $\mathbf{s}u \in \mathbf{B}(\mathbf{p}) \subseteq \mathbf{X}$, and therefore $\mathcal{Q}(\mathbf{s}u) \in \mathbf{B}(\mathcal{Q}(\mathbf{s}))$. This means that $\mathcal{Q}(\mathbf{s})$ is a prefix of $\mathcal{Q}(\mathbf{s}u)$.

We now show that $\text{Repeat}(K) = \mathcal{Q}$ by showing that they have the same prefixes. By construction, $\text{Repeat}(K)$ coincides with $\mathcal{Q}$ on finite streams. Let $\mathbf{s} = s_1 s_2 \dots$ be an infinite stream and let $v_1 \dots v_j$ be an arbitrary prefix of $\text{Repeat}(K)(\mathbf{s})$. Let i be the smallest natural number such that $v_1 \dots v_j$ is a prefix of $\text{Repeat}(K)(s_1 \dots s_i) = \mathcal{Q}(s_1 \dots s_i)$. Since $\mathcal{Q}(\mathbf{s}) \in \mathbf{B}(\mathcal{Q}(s_1 \dots s_i))$, we have $v_1 \dots v_j$ also as a prefix of $\mathcal{Q}(\mathbf{s})$. We conclude that every prefix of $\text{Repeat}(K)(\mathbf{s})$ is also a prefix of $\mathcal{Q}(\mathbf{s})$.

For the other direction, let $v_1 \dots v_j$ be an arbitrary prefix of $\mathcal{Q}(\mathbf{s})$. By continuity, $v_1 \dots v_j$ is also a prefix of $\mathcal{Q}(s_1 \dots s_i)$ for some i. Since $\text{Repeat}(K)(s_1 \dots s_i) = \mathcal{Q}(s_1 \dots s_i)$, we have $v_1 \dots v_j$ also as a prefix of $\text{Repeat}(K)(s_1 \dots s_i)$, which by construction is itself a prefix of $\text{Repeat}(K)(\mathbf{s})$, as desired. □

Theorem 9 can be used to prove that there are simple stream queries that are not abstract computable.

Example 10. Consider the following query CHECK. Let $a, b \in \mathbb{U}$ and let $\mathbf{s}$ be a stream over $\mathbb{U}$. Then $\text{CHECK}(\mathbf{s})$ is the stream (a) if b does not occur in $\mathbf{s}$; otherwise, $\text{CHECK}(\mathbf{s})$ is the empty stream $()$. This query is not abstract computable; we prove that CHECK is not continuous. Consider the open ball $\mathbf{B}(a)$. Clearly, the empty stream $()$ is in $\text{CHECK}^{-1}(\mathbf{B}(a))$. The only open ball that contains the empty stream is $\mathbf{B}(())$. This open ball, however, is not included into $\text{CHECK}^{-1}(\mathbf{B}(a))$. Indeed, $(b) \in \mathbf{B}(())$, but $\text{CHECK}(b) = () \notin \mathbf{B}(a)$. □

Remark 11. In connection to Theorem 9 we remark the following:

1. Suppose we would have extended the Cantor metric to finite (as well as infinite) streams in the obvious manner; in particular, if $\mathbf{s}$ is a finite prefix of $\mathbf{s}'$, but $\mathbf{s} \neq \mathbf{s}'$, then we define $d(\mathbf{s}, \mathbf{s}') = 2^{-(n+1)}$ with n the length of $\mathbf{s}$. In the resulting topology, abstract computable queries need no longer be continuous. A simple example is provided by the query $\mathcal{Q}$ from Example 4.

Let $\mathbb{U} := \{a, b\}$ and let P be true of a and false of b. Consider the open ball $\mathbf{B}$ containing only the empty stream (). Then $\mathcal{Q}$ maps the infinite stream $\mathbf{b}$ containing only b's into $\mathbf{B}$. Any open ball $\mathbf{B}(\mathbf{p})$ around $\mathbf{b}$, however, contains the stream $\mathbf{p}a$ which is not in $\mathcal{Q}^{-1}(\mathbf{B})$. Thus, $\mathcal{Q}$ is not continuous.

2. The qualification in Theorem 9 that $\mathcal{Q}$ must map finite inputs to finite outputs is important for the if-direction. Indeed, any constant query, that always outputs some fixed infinite stream, is continuous, but not abstract computable (precisely because it maps finite to infinite).

4 The Finite Case

Considering only finite streams makes the situation simpler. Define a finite stream query as a mapping from *finStream* to *finStream*. Define abstract computability of finite stream queries in the same way as for queries on *Stream*, and consider the topology on *finStream* induced by the topology on *Stream*, i.e., the open balls are now *finite* continuations of finite streams. We will use the notation $\mathbf{B}_{\text{fin}}(\mathbf{p})$ to denote the set of all finite continuations of the finite stream $\mathbf{p}$. We then indeed have:

Proposition 12. *A finite stream query is abstract computable if and only if it is continuous.*

In the finite case, there is also a third equivalent notion: monotonicity. A query $\mathcal{Q}$: *finStream* $\to$ *finStream* is called monotone if for all $\mathbf{s}, \mathbf{s}' \in$ *finStream*, $\mathbf{s} \sqsubseteq \mathbf{s}'$ implies $\mathcal{Q}(\mathbf{s}) \sqsubseteq \mathcal{Q}(\mathbf{s}')$, where $\sqsubseteq$ denotes the "prefix of" relation.

Proposition 13. *A finite stream query is continuous if and only if it is monotone.*

Proof. For the if-direction let $\mathcal{Q}$: *finStream* $\to$ *finStream* be monotone. Consider $\mathbf{X} := \mathbf{B}_{\text{fin}}(\mathbf{p})$. Let $\mathbf{s}$ be a stream in $\mathcal{Q}^{-1}(\mathbf{X})$. Then, $\mathbf{s} \in \mathbf{B}_{\text{fin}}(\mathbf{s}) \subseteq \mathcal{Q}^{-1}(\mathbf{X})$. Indeed, $\mathbf{s}' \in \mathbf{B}_{\text{fin}}(\mathbf{s})$ implies $\mathbf{s} \sqsubseteq \mathbf{s}'$, which by monotonicity implies $\mathcal{Q}(\mathbf{s}) \sqsubseteq \mathcal{Q}(\mathbf{s}')$. As $\mathcal{Q}(\mathbf{s})$ has $\mathbf{p}$ as a prefix, $\mathcal{Q}(\mathbf{s}')$ has $\mathbf{p}$ as a prefix too and thus $\mathcal{Q}(\mathbf{s}') \in \mathbf{X}$.

The only-if direction is proved by the argument already used in the proof of the if-direction of Theorem 9, where we showed that K is well-defined. Concretely, let $\mathcal{Q}$: *finStream* $\to$ *finStream* be continuous. Let $\mathbf{s} \sqsubseteq \mathbf{s}'$. Consider $\mathbf{X} := \mathcal{Q}^{-1}(\mathbf{B}_{\text{fin}}(\mathcal{Q}(\mathbf{s})))$. By continuity, $\mathbf{X}$ is a union of open balls. Thus, there must be an open ball $\mathbf{B}_{\text{fin}}(\mathbf{p})$ with $\mathbf{s} \in \mathbf{B}_{\text{fin}}(\mathbf{p}) \subseteq \mathbf{X}$. Clearly, $\mathbf{p}$ must be a prefix of $\mathbf{s}$. But then also $\mathbf{s}' \in \mathbf{B}_{\text{fin}}(\mathbf{p}) \subseteq \mathbf{X}$, and therefore $\mathcal{Q}(\mathbf{s}') \in \mathbf{B}_{\text{fin}}(\mathcal{Q}(\mathbf{s}))$. This means that $\mathcal{Q}(\mathbf{s}) \sqsubseteq \mathcal{Q}(\mathbf{s}')$. □

As a corollary we obtain the following equivalence already noted by Law, Wang and Zaniolo (LWZ), who referred to our notion of abstract computability as computability by a "nonblocking" operator:

Corollary 14 ([12]). *Let $\mathcal{Q}$ be a finite stream query. $\mathcal{Q}$ is computable by a nonblocking operator if and only if $\mathcal{Q}$ is monotone.*

The proof given by LWZ is slightly confusing. Their formalism is based on a notion of queries on finite streams that are computable by (not necessarily non-blocking) "operators". They fail to mention, however, that *any* query on finite streams is computable by such an operator.

5 Time

In some applications, the output stream is synchronized with the input stream. In such cases, we need an additional requirement on stream queries beyond mere abstract computability.

Example 15. Consider the following instance of the query from Example 4: the input is a stream of numbers (e.g., sensor readings) and the output consists of all readings below a certain threshold, say 0. In an "untimed" setting, where the original time points of the output readings are not required by the client of the query, we can simply formalize this stream query as being abstract computable with kernel function K_0 with $K_0(()) = ()$, and

$$K_0(\mathbf{s}u) = \begin{cases} u & \text{if } u < 0 \\ () & \text{otherwise.} \end{cases}$$

On the other hand, in a "timed" setting stream positions in the output are supposed to be synchronized with stream positions in the input [3,2]. In that case, the above formalization is inadequate, because, the 5th element of the output may well be, say, the 10th element of the input!

A more proper computation would be given by the function K_1 with again $K_1(()) = ()$, and now

$$K_1(\mathbf{s}u) = \begin{cases} u & \text{if } u < 0 \\ \text{NULL} & \text{otherwise.} \end{cases}$$

where NULL is an explicitly visible element denoting that the reading at this time point was not below 0. □

The above discussion motivates:

Definition 16. *A stream query $\mathcal{Q}$ is synchronous abstract computable (SAC) if $\mathcal{Q} = \text{Repeat}(K)$ for some kernel K: finStream → finStream such that $K(()) = ()$ and every other $K(\mathbf{s})$ is of length one. We will call such kernel K a length-one kernel.*

SAC stream queries can be characterized by means of non-predicting queries. Here and below, $\mathbb{N}_0$ stands for the set of natural numbers without zero.

Definition 17. *A stream query $\mathcal{Q}$ is non-predicting if for all streams $\mathbf{s}$ and $\mathbf{s}'$ and for all $t \in \mathbb{N}_0$ such that $\mathbf{s}^{\leq t} = (\mathbf{s}')^{\leq t}$, we have $\mathcal{Q}(\mathbf{s})_t = \mathcal{Q}(\mathbf{s}')_t$.*

We note that non-predicting is part of the definition of "stream operator" by Arasu, Babu and Widom [3,2].

Proposition 18. *A stream query is SAC if and only if it is non-predicting.*

Proof. Let K be a length-one kernel for stream query $\mathcal{Q}$. Let $\mathbf{s}, \mathbf{s}' \in \mathit{Stream}$ and $t \in \mathbb{N}_0$ such that $\mathbf{s}^{\leq t} = (\mathbf{s}')^{\leq t}$. Then

$$\mathcal{Q}(\mathbf{s})_t = K(\mathbf{s}^{\leq t}) = K((\mathbf{s}')^{\leq t}) = Q(\mathbf{s}')_t$$

and thus $\mathcal{Q}$ is non-predicting.

For the "if" direction, let $\mathcal{Q}$ be non-predicting. For each finite stream $\mathbf{p}$ of length t, define $\pi(\mathbf{p})$ as the infinite stream with $\pi(\mathbf{p})_i = p_i$ for $i \leq t$ and with $\pi(\mathbf{p})_i = p_t$ for $i > t$. Then the following function K is a length-one kernel for $\mathcal{Q}$. If $\mathbf{p}$ is a finite stream of length t then $K(\mathbf{p}) := \mathcal{Q}(\pi(\mathbf{p}))_t$.

Furthermore, for each stream $\mathbf{s}$ and any time instant t, define $\pi'(\mathbf{s}, t)$ as the infinite stream with $\pi'(\mathbf{s}, t)_i = s_i$ for $i \leq t$ and with $\pi'(\mathbf{s}, t)_i = s_t$ for $i > t$. We now prove that K is indeed as desired. Let $\mathbf{s}$ be a stream and let t be a time instant. Then

$$\mathcal{Q}(\mathbf{s})_t = \mathcal{Q}(\pi'(\mathbf{s}, t))_t = \mathcal{Q}(\pi(\mathbf{s}^{\leq t}))_t = K(\mathbf{s}^{\leq t}).$$

Here, the first equality follows from the fact that $\mathcal{Q}$ is non-predicting; the second equality follows from the definition of $\pi'(s, t)$; and the third equality follows from the definition of K. □

We also have:

Proposition 19. *SAC stream queries are closed under composition.*

6 Complexity Limitations

The definition of abstract computability does not impose any restriction on K: the function is not even required to be computable, neither in the classical sense nor in the sense of TTE. The results in the previous sections are thus very general.

To further study the limitations of streaming applications, however, such restrictions are necessary. Concretely, for a class $\mathcal{C}$ of functions from *finStream* to *finStream*, we say that a query $\mathcal{Q}$: *Stream* → *Stream* is *abstract computable modulo* $\mathcal{C}$ if $\mathcal{Q}$ has a kernel K in $\mathcal{C}$. The class $\mathcal{C}$ could for example be the class of functions computable in the classical sense or in the sense of TTE; or—as in the "streaming model of computation" [5]—$\mathcal{C}$ could be the class of functions incrementally computable in polylog space and in polylog time per data element.

In the next section, we will define several classes $\mathcal{C}$ of functions computable by a concrete model based on the Abstract State Machine (ASM) methodology [9], that we will call "streaming ASM" (sASM). We will study abstract computability modulo the classes $\mathcal{C}$ obtained by altering the computation power of the model.

7 Streaming ASMs

An abstract state machine (ASM) is a transition system whose states are many-sorted first-order structures. Transitions change the interpretation of some of the function and relation symbols—those in the *dynamic* part of the vocabulary—and leave the remaining symbols—those in the *static* part of the vocabulary—unchanged. The part of the structure that is never changed during state transitions, i.e., the structure over the static part of the vocabulary, is typically called the *background structure*. Transitions are described by simple rules that produce state updates which are "fired" simultaneously (if they are inconsistent, no update is carried out). A crucial property of the sequential ASM model is that in each transition only a limited part of the state is changed. The detailed definition of sequential ASMs is given in the Lipari guide [9].

We now describe the streaming abstract state machine (sASM) model.

The states: The base set of any state, i.e., the universe of the structure in the sense of logic, contains at least our universe $\mathbb{U}$ of data elements. We assume that $\mathbb{U}$ contains an element $\perp$.

The static functions and predicates on the base set include, but are not limited to, the functions and predicates defined on $\mathbb{U}$.

Each state of an sASM contains a finite number of dynamic functions on the base set. There are always the nullary dynamic function *in* and a number of nullary dynamic functions, called output registers, denoted by *out*, possibly with subscripts. The output registers and *in* take values in $\mathbb{U}$.

The names of the static and dynamic functions and predicates are collected in a vocabulary.

The program: A program for an sASM is a basic sequential program in the sense of ASM theory. Concretely, a basic update rule has the form: $f(t_1, \ldots, t_n) := t_0$ where f is a function name and $t_0, \ldots, t_n$ are terms in the vocabulary. To fire the basic update rule at a state $\mathcal{A}$, evaluate the terms $t_0, \ldots, t_n$ in $\mathcal{A}$ to elements $a_0, \ldots, a_n$ in the base set and then change the interpretation of f in $(a_1, \ldots, a_n)$ to a_0.

Update rules $r_1, \ldots, r_m$ can be combined to a new rule par r_1 ... r_m endpar, the semantics of which is this: Fire rules $r_1, \ldots, r_m$ in parallel; if they are inconsistent then do nothing.

Furthermore, if r_1 and r_2 are rules and φ is a quantifier-free formula in the vocabulary, then if φ then r_1 else r_2 endif is also a rule. The semantics is obvious.

Now, an sASM program is just a single rule.

The run and the output: An sASM M that is set to work on a finite stream $\mathbf{s}$ starts in the state where all dynamic functions have the interpretation $\perp$, except for the function *in*: In the initial state, the function *in* contains the first element of the stream $\mathbf{s}$.

The run of M on $\mathbf{s}$ is the sequence of states obtained as follows: start from the initial state and fire (the rule of) M's program, in each step interpreting

the function *in* as the next element in $\mathbf{s}$. The sASM halts when the end of $\mathbf{s}$ is reached. The interpretation of *in* is dynamic but it is controlled by the environment rather than by the machine; *in* is an *external* function.

We define the *final output of M on a finite stream* $\mathbf{s}$ as the stream obtained by concatenating the interpretations of the output registers in some predefined order when M has halted, and where $\bot$-elements are disregarded.

We now say that an sASM M computes a function $K\colon \mathit{finStream} \to \mathit{finStream}$ (meant as a kernel for a stream query) if for all finite streams $\mathbf{s}$, the final output of M on $\mathbf{s}$ equals $K(\mathbf{s})$. By K_M we denote the function K computed by M.

It is important to note that the final output of an sASM M on a stream $s_1 \dots s_{n+1}$ can be simply obtained by running M on the input $s_1 \dots s_n$ first, and then making one final step upon reading s_{n+1}. Consequently, on any stream $\mathbf{s}$ (finite or infinite), we can compute $\mathrm{Repeat}(K_M)(\mathbf{s})$ simply by continuously running M on $\mathbf{s}$, at each step producing the current output. We refer to $\mathrm{Repeat}(K_M)$ as the *stream query computed by M*.

Example 20. Recall Example 1. In the setting where $\mathbb{U}$ contains sets of entries, there could for example be a function defined on $\mathbb{U}$ that given a set of entries, returns the set of sensor identifiers that measured an alarmingly high value.

In the setting where $\mathbb{U}$ contains entries directly, there could for example be a predicate defined on $\mathbb{U}$ that checks whether an entry has an alarmingly high measurement and a function that given an entry, returns the sensor identifier of the entry. □

Example 21. Consider the sliding window join between two streams of tuples of natural numbers over the attributes $\{A, B\}$ and $\{C, D\}$, where the join condition is $B = C$. The output tuples have attributes $\{A, B, D\}$. The universe $\mathbb{U}$ then contains $\bot$, Tuple_{AB}, Tuple_{CD}, and Tuple_{ABD}, with Tuple_{AB} the set of tuples over the attributes $\{A, B\}$, and similarly for Tuple_{CD} and Tuple_{ABD}. The function $\mathit{join}_{B=C}\colon \mathit{Tuple}_{AB} \times \mathit{Tuple}_{CD} \to \mathit{Tuple}_{ABD}$ checks whether two tuples join on their B- and C-attributes and returns the joined tuple; the result is $\bot$ if the tuples do not join.

The output of the sliding window join depends on two streams, whereas streaming ASMs work on a *single* stream. Moreover, the output depends on the particular interleaving in which the streams arrive. By choosing an appropriate universe $\mathbb{U}$, however, we can represent the two input streams and their interleaving as a single stream.

Concretely, we extend the universe $\mathbb{U}$ with the set *TaggedTuple* of elements of the form $\langle \mathtt{r}\colon u\rangle$ and $\langle \mathtt{s}\colon v\rangle$ with $u \in \mathit{Tuple}_{AB}$ and $v \in \mathit{Tuple}_{CD}$. A tagged tuple encodes an element and its origin. For example, the stream of tagged tuples

$$\langle \mathtt{r}\colon (1,2)\rangle\langle \mathtt{s}\colon (2,3)\rangle\langle \mathtt{s}\colon (3,4)\rangle \dots$$

is a representation of the interleaving of the tuple (1,2) arriving in the first stream, followed by the tuples (2,3) and (3,4) arriving in the second stream, and so on. Furthermore, we add the predicates R and S to the universe $\mathbb{U}$ to test whether an element is of the form $\langle \mathtt{r}\colon u\rangle$ or $\langle \mathtt{s}\colon v\rangle$, respectively. Finally, we add

a function *strip*: $\mathit{TaggedTuple} \to \mathit{Tuple}_{AB} \cup \mathit{Tuple}_{CD}$ that removes the tag of a tagged tuple. Static functions return $\perp$ when one of the arguments is $\perp$.

Assume for simplicity that the window size is 2. We then equip the sASM with 4 nullary dynamic functions reg_i^R, and reg_i^S for $i = 1, 2$. The following is now a program for an sASM computing the sliding window join described above.

```
par
  if R(in) then
    par
      reg_1^R = in
      reg_2^R = reg_1^R
      out_1 = join_{B=C}(strip(in), strip(reg_1^S))
      out_2 = join_{B=C}(strip(in), strip(reg_2^S))
    endpar
  endif
  if S(in) then
    par
      reg_1^S = in
      reg_2^S = reg_1^S
      out_1 = join_{B=C}(strip(reg_1^R), strip(in))
      out_2 = join_{B=C}(strip(reg_2^R), strip(in))
    endpar
  endif
endpar
```

□

8 Bounded-Memory and $o(n)$-Bitstring sASMs

Due to the extreme generality of the ASM model, one should not expect that restricting attention to stream queries that are computable by an sASM would imply any limitation. Indeed, the only restriction that comes from our sASM model is that at each step in the computation of the stream query, only a constant number of elements can be output. More concretely, since the background structure of an sASM could, a priori, be anything, we have the following proposition and corollary (which in itself are philosophically entirely uninteresting):

Proposition 22. *Let k be a fixed natural number and let K: finStream $\to$ finStream be any kernel function such that the length of $K(\mathbf{s})$, for any finite stream $\mathbf{s}$, is at most k. Then the stream query* Repeat(K) *is computable by some sASM.*

Proof (sketch). It is an easy matter for an sASM to compute Repeat(K) if it has 1) a background structure containing a) the set of all finite streams *finStream*, b) the append function of sort $\mathit{finStream} \times \mathbb{U} \to \mathit{finStream}$, c) the function K, and d) functions $\mathit{element}_i$ for $i = 1, \ldots, k$ to extract elements out of a finite stream; and 2) a nullary dynamic function s containing at each step the part of the stream that has already arrived.

At each step, the sASM uses the append function to update the dynamic function s; it applies K to the stream s; and it uses the extraction functions $element_i$ to update the output registers. □

Corollary 23. *Every SAC query is abstract computable by an sASM.*

In order to formulate a relevant complexity limitation on stream queries, we propose "bounded-memory sASMs".

Definition 24. *A bounded-memory sASM is an sASM with the following restrictions: 1) no output register can ever be used as an argument to a function; 2) all dynamic functions are nullary; and 3) non-nullary (static) functions can only be applied in rules of the form out* $:= t_0$*, with out an output register and* t_0 *a term over the vocabulary.*

Example 25. The sASM computing the sliding window join in Example 21 is a bounded-memory sASM. The obvious sASM for computing the running average query from Example 3, however, is not bounded-memory (but see later, when we introduce bitstring sASMs). □

Every CQL-query where a finite window is applied to the input streams ([2]) is computable by a bounded-memory sASM. Indeed, let $\mathcal{Q}$ be such a CQL-query. Then, $\mathcal{Q} = \text{Repeat}(K_M)$, where M is the following sASM. For each window of $\mathcal{Q}$ of size n, the sASM M has n dynamic constants. When M receives a new input element, say with tag $\langle \mathtt{r}: \rangle$, the sASM simulates the sliding of the window(s) on input stream $\mathbf{r}$ by updating the corresponding dynamic constants accordingly. In each step, the output is computed in a brute-force way. This technique was already illustrated in Example 21.

Moreover, every duplicate-eliminating SPJ-query computable in bounded memory in the sense defined by Arasu et al. is computable by a bounded-memory sASM [1].

Bounded-memory sASMs also have some limitations: even the very simple stream query that checks whether two streams intersect, is not computable by a bounded-memory sASM. Let $\mathbb{E}$ be an infinite set of data elements and let *TaggedElement* be the set of elements of the form $\langle \mathtt{r}:u \rangle$ and $\langle \mathtt{s}:u \rangle$ with $u \in \mathbb{E}$. A stream over *TaggedElement* then represents the interleaving of two streams over $\mathbb{E}$ (see Example 21). Let $\mathbb{U}$ be the set *TaggedElement* extended with the boolean values `true` and `false`. The query INTERSECT is defined on streams over $\mathbb{U}$ and checks whether a common element has been seen in the interleaved streams over $\mathbb{E}$. Concretely, the result of INTERSECT on a stream $\mathbf{s}$ over $\mathbb{U}$ is the stream $\mathbf{s}'$ such that the n-th element of $\mathbf{s}'$ is `true` if and only if for some $i, j \in \mathbb{N}_0$ with $i, j < n$ and for some $u \in \mathbb{E}$, we have $s_i = \langle \mathtt{r}:u \rangle$ and $s_j = \langle \mathtt{s}:u \rangle$.

Proposition 26. INTERSECT *is not computable by a bounded-memory sASM.*

Proof. Let M be a bounded-memory sASM such that INTERSECT is equal to $\text{Repeat}(K_M)$.

Let Ω be the set of predicates of M. Then for each predicate $p \in \Omega$ of arity k and for each k-sequence α of elements in $\{\mathtt{r}, \mathtt{s}\}$, define the predicate p^α on $\mathbb{E}$ to be true of a tuple $(u_1, \ldots, u_k)$ iff p is true of $(\langle \alpha_1 : u_1 \rangle, \ldots, \langle \alpha_k : u_k \rangle)$. Let $\Omega' := \{p^\alpha \mid p \in \Omega \text{ and } \alpha \in \{\mathtt{r}, \mathtt{s}\}^k \text{ where } k = \text{arity}(p)\}$.

Without loss of generality, we assume that $\mathbb{E}$ is totally ordered by a predicate $<$. Using Ramsey's theorem, we can find an infinite set $V \subseteq \mathbb{E}$ over which the truth of the predicates in Ω' on tuples of elements in $\mathbb{E}$ only depends on the way these data elements compare w.r.t. $<$ (details on this can be found, e.g., in Libkin's textbook [13, Section 13.3]). Now choose $2n$ elements in V, for n large enough, satisfying $v_1 < v'_1 < \cdots < v_n < v'_n$. Let $\mathbf{s}$ be the input stream $\langle \mathtt{r} : v_1 \rangle \ldots \langle \mathtt{r} : v_n \rangle$ and consider the run of M on $\mathbf{s}$. After the step where $\langle \mathtt{r} : v_n \rangle$ is processed there will be at least one element $\langle \mathtt{r} : v_\ell \rangle$ that M has not stored in its registers. Then, consider the streams $\mathbf{s}'$ and $\mathbf{s}''$ of length $n+1$ that have $\mathbf{s}$ as a prefix, and with $s'_{n+1} = \langle \mathtt{s} : v_\ell \rangle$ and $s''_{n+1} = \langle \mathtt{s} : v'_\ell \rangle$. The runs of M on $\mathbf{s}'$ and $\mathbf{s}''$ will be identical to the run of M on $\mathbf{s}$ until right after the step where $\langle \mathtt{r} : v_n \rangle$ is processed. In the next step, the machine receives either $\langle \mathtt{s} : v_\ell \rangle$ or $\langle \mathtt{s} : v'_\ell \rangle$. Because v_ℓ and v'_ℓ have the same relative order with respect to the other v-elements, each tuple of elements from the set $\{v_1, \ldots, v_\ell, \ldots, v_m\}$ satisfies the same predicates in Ω' as the tuple obtained by replacing v_ℓ by v'_ℓ. By definition of Ω', also each tuple of elements from the set $\{\langle \mathtt{r} : v_1 \rangle, \ldots, \langle \mathtt{s} : v_\ell \rangle, \ldots, \langle \mathtt{r} : v_m \rangle\}$ satisfies the same predicates in Ω as the tuple obtained by replacing $\langle \mathtt{s} : v_\ell \rangle$ by $\langle \mathtt{s} : v'_\ell \rangle$. Therefore, the output of M on $\mathbf{s}'$ will be identical to the output of M on $\mathbf{s}''$. As a consequence, $\text{Repeat}(K_M)(\mathbf{s}')$ and $\text{Repeat}(K_M)(\mathbf{s}'')$ are equal while $\text{INTERSECT}(\mathbf{s}')$ and $\text{INTERSECT}(\mathbf{s}'')$ are different. Thus, M is wrong. □

This result can also be obtained via a reduction from a result on Finite Cursor Machines (FCMs) in our earlier work with Grohe, Schweikardt and Tyszkiewicz [8]. An FCM works by moving one-way cursors over a number of input lists using an internal memory consisting of a finite number of modes, finitely many element registers containing input elements, and finitely many registers containing bitstrings. To manipulate its internal memory, an FCM has a number of functions and predicates, with the restriction that the output of a function is always a bitstring. It has been shown [8, Theorem 12] that no matter how rich the background is, an FCM can not check whether two sets intersect using bitstring registers of size $o(n)$, where n is the size of the input.

The proof we gave here is more direct and therefore provides more insight on the limitations of bounded memory stream processing. The reduction argument, however, can easily be generalized to accommodate for bitstring registers of size $o(n)$. A *bitstring sASM* is an sASM defined as in Definition 24 with the following relaxation of restriction 3: non-nullary (static) functions can be used also to update non-output registers, as long as those functions produce bitstrings. An $o(n)$-sASM then is a bitstring sASM such that on each stream $\mathbf{s}$ and for each step n in the run on $\mathbf{s}$, the sASM stores bitstrings of length $o(n)$.

Example 27. We can model a version of the running average query (Example 3) using $o(n)$-bitstring sASMs. Indeed, consider streams of natural numbers such that the value in the n-th position of the stream (for any n) is at most $2^{\text{polylog}(n)}$.

Then with a static function from natural numbers to their binary representations, and the addition and division function on binary numbers, we can compute the running average with an $o(n)$-sASM. □

Proposition 28. *The query* INTERSECT *is not computable by an $o(n)$-sASM.*

Proof. Let M be an $o(n)$-sASM M working on a stream of tagged elements such that INTERSECT is equal to Repeat(K_M). From M, we can then construct an $o(n)$-FCM M' working on two lists of elements in $\mathbb{E}$ that checks whether they have a common element. The FCM M' has the same number of bitstring registers as M, and has an element register for every dynamic constant of M. For every element in an element register, M' remembers from which input list the element was copied, using its internal mode. Furthermore, let Ω be the set of predicates of M, including the predicates naturally corresponding to M's boolean output functions. Then the set of predicates of M' is the set Ω' as defined in the proof of Proposition 26. Finally, if $\mathcal{F}$ is the set of functions of M, then the set of functions $\mathcal{F}'$ of M' is similarly constructed from $\mathcal{F}$ as Ω' is constructed from Ω.

Consider the input lists R and S. The FCM M' has a single cursor on R and a single cursor on S. Now, M' computes as follows. At each odd step, M' moves its cursor on R to the next element u, updating the (element and bitstring) registers as M would do when receiving the element $\langle \mathtt{r}:u\rangle$ from the stream. The internal mode is changed so that it contains the origin of each element in the registers. At each even step, M' moves its cursor on S to the next element v, updating the registers as M would do when receiving the element $\langle \mathtt{s}:v\rangle$ from the stream. The internal mode is again changed accordingly. M' can simulate this behaviour using the functions in $\mathcal{F}'$, or the predicates in Ω' together with its internal mode. For example, if M applies a predicate p to an element in a dynamic constant *reg* — i.e., an element of the form $\langle \mathtt{r}:u\rangle$ or $\langle \mathtt{s}:v\rangle$ — the FCM M' would use its internal mode to obtain the origin of the element in the register corresponding to *reg* and then apply the right predicate p^r or p^s to the element in that register — i.e., to u or v. Once M outputs true, M' enters the accept state and halts. As long as M outputs false, M' continues until it has detected the ends of the input lists. In that case, M' enters the reject state and halts. Note that M' can use the predicates corresponding to the boolean functions of M to obtain the output M produces. Because M works correctly, it will also work correctly on this particular interleaving. Therefore, M' correctly checks whether R and S intersect. Hence the contradiction. □

We conclude by pointing out that on finite streams, finite cursor machines are indeed more powerful than bounded-memory sASMs: Consider the query SORT-INTERSECT that given two finite streams A and B, checks if they are both sorted and if so, outputs their intersection; if the inputs are not sorted, SORT-INTERSECT, outputs false. Then,

Proposition 29. *The query* SORT-INTERSECT *is computable by an FCM but not by a bounded-memory sASM.*

Proof. An FCM would compute the query SORT-INTERSECT using one cursor on each list to check if they are sorted and another cursor on each list to do a synchronized scan of both list to search for common elements. Inspection of the proof of Proposition 26 reveals that a bounded-memory sASM can not even check whether two finite sorted streams intersect. □

9 Conclusion

An interesting open problem is to relax the definition of bounded-memory sASM in other ways than with using $o(n)$-length bitstrings.

References

1. Arasu, A., Babcock, B., Babu, S., McAlister, J., Widom, J.: Characterizing memory requirements for queries over continuous data streams. ACM TODS 29(1), 162–194 (2004), Includes an electronic appendix
`http://doi.acm.org/10.1145/974750.974756`
2. Arasu, A., Babu, S., Widom, J.: The CQL continuous query language: semantic foundations and query execution. The VLDB Journal 15(2), 121–142 (2006)
3. Arasu, A., Widom, J.: A denotational semantics for continuous queries over streams and relations. SIGMOD Record 33(3), 6–11 (2004)
4. Ayres, F., Mendelson, E.: Schaum's Outline of Calculus. McGraw-Hill, New York (1999)
5. Babcock, B., Babu, S., Datar, M., Motwani, R., Widom, J.: Models and issues in data stream systems. In: PODS 2002, pp. 1–16 (2002)
6. Balakrishan, H., Balazinska, M., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Galvez, E.F., Salz, J., Stonebraker, M., Tatbul, N., Tibbetts, R., Zdonik, S.B.: Retrospective on Aurora. The VLDB Journal 13(4), 370–383 (2004)
7. Golab, L., Özsu, M.T.: Processing sliding window multi-joins in continuous queries over data streams. In: Aberer, K., Koubarakis, M., Kalogeraki, V. (eds.) Databases, Information Systems, and Peer-to-Peer Computing. LNCS, vol. 2944, pp. 500–511. Springer, Heidelberg (2004)
8. Grohe, M., Gurevich, Y., Leinders, D., Schweikardt, N., Tyszkiewicz, J., Van den Bussche, J.: Database query processing using finite cursor machines. In: Schwentick, T., Suciu, D. (eds.) ICDT 2007. LNCS, vol. 4353, pp. 284–298. Springer, Heidelberg (2006)
9. Gurevich, Y.: Evolving algebra 1993: Lipari guide. In: Börger, E. (ed.) Specification and Validation Methods, pp. 9–36. Oxford University Press, Oxford (1995)
10. Gurevich, Y.: Sequential abstract-state machines capture sequential algorithms. ACM TOCL 1(1), 77–111 (2000)
11. Hocking, J.G., Young, G.S.: Topology. Dover Publications, Mineola, NY (1988)
12. Law, Y.-N., Wang, H., Zaniolo, C.: Query languages and data models for database sequences and data streams. In: VLDB 2004, pp. 492–503 (2004)
13. Libkin, L.: Elements of Finite Model Theory. Springer, Heidelberg (2004)
14. Madden, S.R., Franklin, M.J., Hellerstein, J.M., Hong, W.: TinyDB: An acquisitional query processing system for sensor networks. ACM TODS 30(1), 122–173 (2005)
15. Terry, D., Goldberg, D., Nichols, D., Oki, B.: Continuous queries over append-only databases. In: SIGMOD 1992, pp. 321–330 (1992)
16. Weihrauch, K.: Computable Analysis: An Introduction. Springer, Heidelberg (2000)