EFFICIENT COMPUTATION OF QUERIES ON FEATURE STREAMS

Simone Santini*

Escuela Politécnica Superior
Universidad Autónoma de Madrid

Abstract

This paper introduces the notion of virtual feature stream, a feature stream defined from a primary data stream, in which at any time only the features that are needed to compute the queries that are currently running in the system are computed.

Virtual feature streams are, in general, impossible to determine a priori, but the paper introduces an algorithm that stops the computation of features as soon as it can be proved that they are no longer needed thus generating, albeit in a roundabout and more expensive than the ideal way, a feature stream that is less expensive than the complete one to compute and safe: the queries that accept the virtual feature stream are those (and only those) that would accept the original feature stream.

1 Introduction

The contemporary development of data base systems is characterized by an effort to expand their reach by giving them the possibility to store and query data types different than those of traditional data bases. Of particular interest for multimedia are methods for querying data that come in the form of streams [13, 18]. The following section will give a precise definition of stream but, roughly, a stream is an infinite list of homogeneous data: in video, these data are images [17], in numeric streams they are numbers or structures containing numbers, and so on. The organization of a stream processing system poses new challenges to data base research because its organization is in a sense dual to that of a traditional data base system: a standard data base is characterized by a permanent data store and transient queries---a query is computed once in a single computation, and then it disappears from the system---while in a stream data base, vis-à-vis the impossibility of storing the data, it is the queries that are permanent and the data that are volatile.

A data base for streams will contain, at any given moment, a set of queries, waiting to be answered, and a set of query executions: queries that have met some of the conditions for their satisfaction, but that are waiting for further data in order to complete their execution and return a result [5]. Stream data bases dealt originally

*This work was supported in part by Consejería de Educación de la Comunidad Autónoma de Madrid, under the grant CCG08-UAM/TIC/4303, Búsqueda basada en contexto como alternativa semántica al modelo ontológico and by the project Ramón y Cajal of the Ministerio de Educación y Ciencia.
with relatively simple streams, such as numerical or tuple streams [15, 5]. In recent years, there have been a number of extensions and domain enlargements, two of which deserve some mention here: on the one hand, to streams of tree-structured elements [6, 14], on the other hand to multimedia streams, such as video and audio. In the area of tree streams, some techniques have been developed that can be a useful complement to those presented here, such as methods for translating tree pattern queries into finite state automata, or to merge the common parts of finite state automata [14, 11], a technique that can be employed as a preliminary (compile-time) preparation for the (run-time) optimization presented here. Multimedia data bases differ from other stream data bases in that the queries do not operate directly on the incoming data; rather, a suitable number of features are computed, and used to evaluate the queries. These features may also form streams, which Liu, Gupta and Jain call feature streams [19]. Computing features in a multimedia system is an expensive operation: so much that it can easily become the bottleneck of the query evaluation, an observation that calls for a shift in the focus of query optimization with respect to data base systems that deal with simpler data types [1].

In non-multimedia stream systems the primary concerns are operator optimization [8, 22], memory occupation [21, 3, 4], elimination of duplicate computation [11] or other forms of evaluation optimization [2, 24]. In multimedia data bases, however, since features are so expensive to compute, a significant source of optimization can be the elimination of feature computations that are not necessary for the advancement of any active query [16]. In this paper, I shall present algorithms to achieve this form of optimization. The algorithms will consider a collection of feature streams and a query pool in the data base system, and will stop the computation of a feature as soon as they determine that it is no longer necessary for the execution of any query in the pool. The query pool may change but, as will be evident by the algorithm, a new query that enters the system will be answered only after a certain delay, not greater than the longest computation time of the features used by that query. The algorithms are provably correct, in the sense that, although they only compute a fraction of the features available, no query fails to be answered because of this. In the next section, I will give a definition of streams and prove some properties that will be used in the rest of the paper. Section 3 will present a query evaluation algorithm, and prove its correctness, in the hypothesis that all features are instantaneous, that is, the value of a feature at time $t$ depends only on the value of the input stream at time $t$. Section 4 will extend the technique to features that depend on several past values of the stream. (a finite number of them), present the relative algorithm, and prove its correctness. This will allow us to cover features that are conceptually multi-element (such as those characterizing motion [12] or long-term trends [25]) as well as those that are conceptually instantaneous, but whose computation is based on a number of consecutive frames for the sake of insensitivity to noise. Section 5 will present some evaluation of the technique, determining how many feature computations it saves.

2 Streams

Many papers on video data bases have a somewhat cavalier attitude towards the definition of basic concepts such as that of stream. This situation is bound to generate problems later on because, in the absence of properly defined concepts, it is difficult to impose the conditions that one might need, such as homogeneity (especially vis-à-vis missing features) or the causality of the feature extraction functions.

There are, grosso modo two possible ways of looking at a stream: either as a
function that returns a different value each time it is called, or as an infinite list of values (that is, as a function from natural numbers to values). The first way is most convenient if one is modeling programs, since it is quite a natural way to model buffered streams; the second is most handy to prove global properties of a stream.

Streams are always streams of something, that is, they are composite data types built on a given data typing system. It will be convenient to assume that the type system on which we build the stream contain the type ⊥ (the data type with a single value). The first definition of stream can be traced back to [7].

Definition 2.1. A stream of elements of type \( \alpha \), that is, a datum of type \( \Sigma(\alpha) \) is a function

\[
\pi : \Sigma(\alpha) \equiv \bot \rightarrow \alpha \times \Sigma(\alpha)
\]

That is, a stream is a function that takes no values and returns a pair: the first element of the pair is the current value of the stream; the second is also a stream: the stream that will return the rest of the values. The second definition of a stream is a function that associates a value to each positive time instant.

Definition 2.2. An \( f \)-stream (from function) of elements of type \( \alpha \), that is, a datum of type \( M(\alpha) \), is a function

\[
q : M(\alpha) \equiv \mathbb{N} \rightarrow \alpha
\]

The first definition is convenient when we are modeling programs, since it gives a natural way to model the passage of time implicitly without having to introduce external counters: every time we read a stream and replace the stream variable with the second element of the value that it returns we are stepping forward in time. The model is simply that of a buffered stream, with the proviso that, in the functional framework in which we are writing the algorithms, the buffer resulting from reading an element has to be returned as part of the result.

This definition is convenient because, just as in buffered streams, there is no way to "skip" time steps: there is no other way of accessing the element \( t+1 \) but reading the element \( t \) first.

The second definition allows a random access, so to speak, to the values: one can read them in any order. For this reason, this definition is not a good model of program behavior, but it is very convenient when we need to prove properties of streams and algorithms. The algorithms that we shall see in this paper are written using the standard definition of stream, that is, the first definition above. However, the proofs of the correctness theorems are technically simpler if we use the second definition, that of \( f \)-streams. For the proof to make logical sense, then, it is necessary to show that streams and \( f \)-streams---the programming construct and its theorem-proving doppelgänger---are one and the same. It should be stressed again that the concept of \( f \)-stream, is not necessary in the programming practice, since in that case the operative definition is the first one, but that \( f \)-streams and the isomorphism theorem will give us powerful mathematical tools to prove the properties of the algorithms that we are about to write.

2.1 Isomorphism of streams and \( f \)-streams

Let us begin by analyzing the first definition. Given a pair \((x,y)\), I shall indicate with \( \pi_1 \) and \( \pi_2 \) the natural projections on the two components of the pair, that is,
Given a stream $\pi : \Sigma(\alpha)$, the value of $\pi$ at time 0 is $(\pi_1 \circ \pi)(())$, while the value of $\pi$ at time $k+1$ is the value of $\pi(\pi)$ at time $k$. It is easy to infer that the value of $\pi$ at time $k \geq 0$ is

$$(\pi_1 \circ \pi_2^k \circ \pi)(()).$$

(3)

Two streams $\pi, \eta : \Sigma(\alpha)$ are equal if they have the same value at all times:

$$\pi = \eta \iff \forall k \geq 0 \quad (\pi_1 \circ \pi_2^k \circ \pi)(()) = (\pi_1 \circ \pi_2^k \circ \eta)(())$$

(4)

This is equivalent to the recursive definition

$$\pi = \eta \iff \pi_1(\pi) = \pi_1(\eta) \land \pi_2(\pi) = \pi_2(\eta)$$

(5)

In order to determine the relation between streams and f-streams I shall define two functions that transform one into the other. The function $F : \Sigma(\alpha) \rightarrow M(\alpha)$ is defined as

$$F[\pi] = \lambda k.(\text{if } k = 0 \text{ then } \pi_1(\pi)() \text{ else } F[\pi_2](k-1))$$

(6)

while the function $G : M(\alpha) \rightarrow \Sigma(\alpha)$ is defined as

$$G[q] = \lambda.(q(0),G[\lambda u. q(u+1)])$$

(7)

The first lemma, whose proof is in appendix, will tell us that the function $F[\pi]$ computes the same values as the stream $\pi$.

Lemma 2.1. For each $k \geq 0$ and $\pi : \Sigma(\alpha)$ we have:

$$F[\pi](k) = (\pi_1 \circ \pi_2^k \circ \pi)(())$$

(8)

The following lemma gives part of the relation between $F$ and $G$. Its proof is also in appendix.

Lemma 2.2. For each $q : M(\alpha)$ we have $F[G[q]] = q$.

In order to prove the dual theorem, I need a technical intermediate result:

Lemma 2.3. For each $\pi, \eta : \Sigma(\alpha)$ the following property holds:

$$\eta = \pi_2 \circ \pi \iff F[\eta] = \lambda u. F[\pi](u+1)$$

(9)

Proof. I shall prove the lemma by showing that the equality of the values of the two functions for a generic $k$ is equivalent to the condition on the left in the theorem statement.

$$(\lambda u. F[\pi](u+1))(k) = F[\pi](s+1) = F[\pi_2 \circ \pi] \quad \text{(by definition of } F)$$

(10)

Theorem 2.1. For all $\pi : \Sigma(\alpha)$ it is

$$G[F[\pi]] = \pi$$

(11)
Proof. I shall use the equality criterion for streams (4) and prove by induction that
\[ \forall k \geq 0 \quad (\pi_1 \circ \pi_2^k \circ G[F[x]])(\alpha) = (\pi_1 \circ \pi_2^k \circ \tau)(\alpha). \] (12)

For \( k = 0 \) we have
\begin{align*}
\pi_1(G[F[x]]) &= F[x](0) \quad \text{(by definition of } G) \\
&= (\pi_1 \circ \tau)(\alpha) \quad \text{(by definition of } F) \tag{13}
\end{align*}

Suppose now that the theorem is true for \( k \). We can write
\[ (\pi_1 \circ \pi_2^{k+1} \circ G[F[x]])(\alpha) = (\pi_1 \circ \pi_2^k \circ (\pi_2 \circ G[F[x]]))(\alpha) \tag{14} \]

Consider now the quantity \( \pi_2 \circ G[F[x]] \). We have
\begin{align*}
\pi_2(G[F[x]]) &= G[\lambda u.F[x](u + 1)] \quad \text{(by definition of } G) \\
&= G[F[\pi_2 \circ \tau]] \quad \text{(by lemma 2.3)} \tag{15}
\end{align*}

replacing this in the second term of (14) one obtains
\begin{align*}
(\pi_1 \circ \pi_2^k \circ (\pi_2 \circ G[F[x]]))(\alpha) &= (\pi_1 \circ \pi_2^k \circ G[F[\pi_2 \circ \tau]])(\alpha) \\
&= (\pi_1 \circ \pi_2^k \circ \pi_2 \circ \tau) \quad \text{(by the inductive hypothesis)} \\
&= (\pi_1 \circ \pi_2^{k+1} \circ \tau) \tag{16}
\end{align*}

Theorems 2.2 and 2.1 show that \( G \) is the inverse of \( F \), so from now on I shall write it \( F^{-1} \). Its existence proves the following theorem:

Theorem 2.2. \( \Sigma(\alpha) \) and \( M(\alpha) \) are isomorphic.

This isomorphism gives us the necessary identification of data types: streams, as defined in definition 2.1 are really the same thing as functions from the set of natural numbers to the data type of the stream elements. Consequently, from now on, I shall generally use the symbol \( \Sigma(\alpha) \) to indicate both data types, except when the distinction between the two is important.

2.2 Causal functions on streams

In multimedia, as we shall see shortly, one is particularly interested in functions that transform a stream (the primary stream) into another one (the derived stream). There can be two flavors of these functions, depending on whether one works with streams or f-streams. However, the functions \( F \) and \( F^{-1} \) implicitly define functionals that transform one type of functions into the other.

Definition 2.3. Given the function \( g : \Sigma(\alpha) \rightarrow \Sigma(\beta) \), the function
\[ F^*[g] : M(\alpha) \rightarrow M(\beta) \] (17)
is defined, for \( q : M(\alpha) \), as
\[ F^*[g](q) = F[g(F^{-1}[q])] \] (18)
Definition 2.4. Given the function \( h : M(\alpha) \to M(\beta) \) the function

\[
F_*[h] : \Sigma(\alpha) \to \Sigma(\beta)
\]

is defined, for \( \mathfrak{r} : \Sigma(\alpha) \) as

\[
F_*[h](\mathfrak{r}) = F^{-1}[h(F[\mathfrak{r}])]
\]

(19)

It is easy to see that \( F^*[F_*[h]] = h \) and that \( F_*[F^*[g]] = g \).

We are interested in functions that, at any given time, can be computed based on the stream values that have already arrived, that is, in causal functions. In order to clarify the concept of causality, I need to define an equivalence, on which causality is based.

Definition 2.5. Two f-streams \( p, q : M(\alpha) \) are \( t \)-equivalent, written \( p \approx t q \), if

\[
\forall \tau \in \mathbb{N} \quad \tau \leq t \Rightarrow p(\tau) = q(\tau).
\]

Lemma 2.4. If \( p \approx t q \), then \( \forall \tau \quad \tau \leq t \Rightarrow p \approx \tau q \).

(The proof is a trivial application of the definition.)

Definition 2.6. A function \( f : M(\alpha) \to M(\beta) \) is \( t \)-causal if

\[
\forall p, q : M(\alpha) \quad p \approx t q \Rightarrow f(p) \approx t f(q).
\]

(21)

Definition 2.7. A function \( f : M(\alpha) \to M(\beta) \) is causal if for all \( t \in \mathbb{N} \) the function \( \lambda x.f(x)(t) : M(\alpha) \to \beta \) is \( t \)-causal.

It is easy to show that a function is causal if and only if for every \( t \), \( h(p)(t) \) does not depend on \( p(t') \) for any \( t' > t \). In particular, I shall call a function strongly causal if it is of the form \( h(p)(t) = \hat{h}(p(t)) \), with \( \hat{h} : \alpha \to \beta \).

This definition of causality is limited to functions on f-streams, but the functional \( F^* \) gives us an obvious way to extend it to functions on streams.

Definition 2.8. A function \( g : \Sigma(\alpha) \to \Sigma(\beta) \) is causal (resp. strongly causal) if \( F^*[g] \) is causal (resp. strongly causal).

One can derive a rich theory of causal functions on streams, but much of it is beyond the scope of this paper. I shall derive just a few results that are useful in the present situation.

Definition 2.9. A function \( g : \Sigma(\alpha) \to \Sigma(\beta) \) is a homomorphism if it is of the form

\[
g(\mathfrak{r}) = (\hat{g}(\pi_1 \circ \mathfrak{r})()), g(\pi_2 \circ \mathfrak{r}))
\]

with \( \hat{g} : \alpha \to \beta \).

Theorem 2.3. Any homomorphism \( g : \Sigma(\alpha) \to \Sigma(\beta) \) is strongly causal.

Proof. I shall prove by induction on \( k \) that, for an arbitrary f-stream \( p \) we have

\[
\forall k \geq 0 \quad F^*[g](p)(k) = \hat{g}(p(k))
\]

(23)

where \( \hat{g} \) is the function that appears in the definition of the homomorphism.
For $k = 0$ we have
\[
F^*[g](p)(0) = F[g(F^{-1}(p))](0) \quad \text{(definition of } F^*)
\]
\[
= \pi_1(g(F^{-1}(p))) \quad \text{(by lemma 2.1)}
\]
\[
= \hat{g}(\pi_1(F^{-1}(p))) \quad \text{(by definition of homomorphism)}
\]
\[
= \hat{g}(p(0)) \quad \text{(by definition of } F^{-1})
\]
(24)

In order to prove the second part, I shall use the following equality:
\[
\pi_2(g(F^{-1}(p))) = g(\pi_2(F^{-1}(p))) = g(F^{-1}[\lambda u.p(u + 1)])
\]
(25)
where the first equality comes from the definition of homomorphism and the second from the definition of $F^{-1}$.

Suppose now that the theorem is true for $k$:
\[
F^*[g](p)(k + 1) = F[g(F^{-1}(p))](k + 1) \quad \text{(definition of } F^*)
\]
\[
= (\pi_1 \circ \pi_2 \circ (\pi_2 \circ g(F^{-1}(p))))(k) \quad \text{(lemma 2.1)}
\]
\[
= (\pi_1 \circ \pi_2 \circ g(F^{-1}[\lambda u.p(u + 1)]))(k) \quad \text{(from 25)}
\]
\[
= F^*[g](\lambda u.p(u + 1))(k) \quad \text{(definition of } F^*)
\]
\[
= \hat{g}((\lambda u.p(u + 1))(k)) \quad \text{(inductive hypothesis)}
\]
\[
= \hat{g}(p(k + 1))
\]
(26)

Strict causality is a fairly restrictive condition, since it requires that each datum of the stream that is being produced depend only on the corresponding datum of the argument. In practice, however, it is easy to see that all the properties proved so far remain valid (with causality in lieu of strict causality) if we allow the function $\hat{g}$ to maintain a state, which allows us to make each element produced by a function depend (in a limited but, pragmatically, general enough way) on the past history of the stream.

Definition 2.10. Let $\gamma$ be a data type, $\hat{g}: \gamma \times \alpha \rightarrow \beta$, and $z: \gamma \times \alpha \rightarrow \gamma$. A homomorphism with state is a function $g: \gamma \rightarrow \Sigma(\alpha) \rightarrow \Sigma(\beta)$ of the form
\[
g(s)(\bar{x})(\bar{y}) = (\hat{g}(s, (\pi_1 \circ \bar{x})), g(z(s, (\pi_1 \circ \bar{x})))(\pi_2 \circ \bar{y}))
\]
(27)

The following theorem can be proved essentially in the same way as the similar property for strict causality:

Theorem 2.4. Let $g$ be a homomorphism with state, with states of type $\gamma$. Then for every $s: \gamma$, $g(s)$ is causal.

If one pictures $\bar{x}$ as a sequence of data arriving at certain points in time, causality implies that $f(\bar{x})$ is also a stream and it can be computed as the data of $\bar{x}$ arrive.

2.3 Feature streams

The importance of stream functions arises from the observation that, in a multimedia information system, one rarely queries the stream directly. Rather, one computes a certain number of features that represent salient characteristics of the stream elements, and creates queries based on them. These features Features computed on
streams also form streams that, following Liu, Gupta and Jain [19] I shall call feature streams. I shall consider two types of feature streams. The first one, which I shall consider in the next section, is composed of features whose computation function is strongly causal (viz. a homomorphism), that is, features whose value at time $t$ depends only on the value of the data stream at time $t$. The second type is computed by homomorphisms with state, with a limitation: I shall assume that the state only contains information about a limited portion of the stream. This implies that there is a number $k$ such that the computation of the feature can be initiated $k$ instants before the feature is used.

I shall apply to the elements of the stream $\mathfrak{f}$ $m$ features extractors $\phi_i : \alpha \rightarrow \beta_i$, which will generate the $m$ streams $\phi_i \mathfrak{f} : \Sigma(\beta_i)$. Whenever useful, I will denote the collection of these streams with the notation $\phi_1 \mathfrak{f} \ldots \phi_m \mathfrak{f} : \Sigma(\beta_1 \times \ldots \times \beta_m) \equiv \Sigma(\beta_1) \times \ldots \times \Sigma(\beta_m)$ (28)

**Example:** In a video stream, one can imagine two features being extracted from the dominant object in every image: its motion, represented by a vector $v \in \mathbb{R}^2$, and its size, that we can consider as represented by the length of the major axis of the object (in pixels) and its area (also in pixels), that is, $S : \mathbb{N} \times \mathbb{N}$.

Given a stream $\mathfrak{f} : \Sigma(\text{img})$, where "img" is the data type of images, the first feature computes a stream $\phi_1 \mathfrak{f} : \Sigma(\mathbb{R}^2)$, which gives, at any time, the speed of the most prominent object in the scene, and the second determines a stream $\psi_1 \mathfrak{f} : \Sigma(\mathbb{N} \times \mathbb{N})$, which gives, at any time, the longest axis and the area of the same object.

The velocity feature is causal, but not strictly causal: it needs to maintain a state with information on at least two consecutive frames in order to determine velocity.

In practical situations, however, the segmentation of the object will not be done based on a single frames, since, when a video is available, segmentation based on motion is much more reliable than segmentation based on a single image [20]. So, in practice, both features will require consideration of a certain number of frames, and neither of them will be strictly causal.

3 Queries on streams

Stream query languages can be defined in a variety of ways. Some are derived from data base query languages, often from the omnipresent SQL [1, 9]. Since streams are sequences of elements, I shall consider here queries expressed in the form of finite state automata. This choice has the advantage of including as a special case the most common query languages: finite state automata on sequences are equivalent to monadic second order logic on the successor relation [23], thereby including first order languages such as SQL and its stream derivates [15]; on trees, they are equivalent to monadic second order logic on the child relation, subsuming most pattern languages. Techniques based on finite state automata are being investigated for the implementation of pattern query languages. There is also a rich literature on the optimization of this representation that, although out of the scope of this paper, makes it an attractive algebra for the translation of many current query languages. In other words, finite state automata are a convenient operational language in which virtually all declarative languages that are being investigated can be translated and optimized, and they are a fairly obvious candidate as the input notation for the algorithms presented here. In this case, of course, being streams sequences of elements we are interested in using finite state automata on sequences. This motivates the following definition:
Definition 3.1. A query $q$ on a stream is a finite state automaton $q = (S, \Delta, s_0, F)$, where $S$ is a finite set, $s_0 \in S$, $F \subseteq S$, and $\Delta$ is a function that can take two forms:

i) $\Delta : S \rightarrow \beta_1 \times \cdots \times \beta_m \rightarrow 2^S$ (nondeterministic query);

ii) $\Delta : S \rightarrow \beta_1 \times \cdots \times \beta_m \rightarrow S$ (deterministic query).

$S$ is called the set of states of the query, $\Delta$ its state transition function, $s_0$ its initial state, and $F$ its set of final states.

Note that I have used the curried form of the state transition function, as this form will come handy in the following.

Definition 3.2. A run at time $t$ (or $t$-run) for a query $q$ on a collection of streams $x_1, \ldots, x_m$ is a sequence $[s_0, s_1, \ldots, s_r]$ (29) such that $s_i \in S$ ($s_0$ being the initial state), and

\[
\forall k \ 0 \leq k < r \Rightarrow s_{k+1} \in \Delta(s_k)(x_1(t+k), \ldots, x_m(t+k)) \quad (30)
\]

if the query is nondeterministic, and

\[
\forall k \ 0 \leq k < r \Rightarrow s_{k+1} = \Delta(s_k)(x_1(t+k), \ldots, x_m(t+k)) \quad (31)
\]

if the query is deterministic.

A $t$-run is called accepting if $s_r \in F$, in which case we say that the query has been satisfied at time $t + r$.

Example: Consider again the previous system of features; one can ask, for instance, the following queries:

$q_1$: an object having an area at least $\alpha$ moves with a velocity vector in the first quadrant for at least three frames;

$q_2$: an object moves with a velocity vector in the third quadrant for at least three frames and its area is no smaller than $\alpha$ except, possibly, during one of the three;

$q_3$: an object (of any area) moves in the third quadrant for at least three frames.

Using the following definitions:

\[
W_1 \equiv (\pi_1 \circ \pi_1 \circ \phi) > 0 \land (\pi_2 \pi_1 \circ \phi) \geq 0 \quad \text{(the object moves in the first quadrant)}
\]

\[
W_3 \equiv (\pi_1 \circ \pi_1 \circ \phi) > 0 \land (\pi_2 \circ \pi_1 \circ \phi) < 0 \quad \text{(the object moves in the third quadrant)}
\]

\[
Z \equiv (\pi_1 \circ \pi_1 \circ \psi) > \alpha \quad \text{(the object has size greater than $\alpha$)}
\]

the three queries are calculated by the following finite state automata (the branches without conditions are traversed when none of the conditions of the other branches are satisfied, $s$ is the starting state, and final states are represented with a double circle):


3.1 $T$-void and safe streams

Given feature streams used to compute a query, we are interested in those occasions in which the computation of one or more features can be avoided without impacting the queries that the system is trying to compute. The following concept will help formalize this notion.

Definition 3.3. Let $r: \Sigma(\alpha)$ be a stream\(^1\), and $T \subseteq \mathbb{N}$. The $T$-void of $r$, $[T]r$, is defined as

$$[T]r(t) = \begin{cases} \bot & \text{if } t \in T \\ r(t) & \text{if } t \notin T \end{cases}$$

Given $m$ features streams $\phi^1_m x$ and $m$ sets $T_i \in \mathbb{N}$, I shall indicate their $T$-voids $[T_1]\phi^1_{i\,1}, \ldots, [T_m]\phi^1_{i\,m}$ as $[T^1_{m\,i}]\phi^1_{i\,m\,x}$. A stream that is the result of a $T$-void for some $T$ is called a virtual stream.

Definition 3.4. Let $r(q) = [s_0, \ldots, s_r]$ be a run for query $q$ at time $t$, $\phi^1_{m\,x}$ the collection of feature streams on which the state transition functions of the query depend, and $[T^1_{m\,i}]\phi^1_{i\,m\,x}$ a $T^1_{m\,i}$-void of $\phi^1_{i\,m\,x}$. $[T^1_{m\,i}]\phi^1_{i\,m\,x}$ is safe for $r(q)$ if, whenever $t + k \in T_i$, $\Delta(s_k)$ does not depend on the feature $\phi_i$.

The general idea of the definition is that from a feature stream $\phi^1_m x$ we can remove those values that are not actually used to compute the state transitions of a run. The following theorem is, given the definition, quite obvious:

Theorem 3.1. Let $r$ be a run at time $t$ and $[T^1_{m\,i}]\phi^1_{i\,m\,x}$ a $T^1_{m\,i}$-void stream that is safe for it. Then $r$ is accepting on $[T^1_{m\,i}]\phi^1_{i\,m\,x}$ if and only if it is accepting on $\phi^1_{i\,m\,x}$.

We can derive a weaker condition if we take into account the typical form of the state transition function $\Delta(s)$. Typically, this function is of the form\(^2\).

$$\Delta(s)(\phi_1, \ldots, \phi_m) =
\begin{align*}
&\text{if } C_1(\phi_{1\,1}, \ldots, \phi_{1\,n_1}) \rightarrow s_1 \\
&\quad \boxed{C_2(\phi_{2\,1}, \ldots, \phi_{2\,n_2}) \rightarrow s_2}
\end{align*}$$

\(^1\)Note that, strictly speaking, I should have considered a stream of type $M(\alpha)$, since in the following equation I consider $r$ as a function. However, as I already mentioned, the isomorphism theorem allows us to identify $M(\alpha)$ and $\Sigma(\alpha)$.

\(^2\)In all the algorithms of this paper, the thick line will denote a non-deterministic order of execution.
Consider a run \( r = [s_0, \ldots, s_r] \) of the query \( q \), the transition \( s_i \rightarrow s_{i+1} \) of that run, and the function

\[
\Delta(s_i) : \beta_1 \times \cdots \times \beta_m \rightarrow S.
\]  

(36)

\[\Delta(s_i) : \beta_1 \times \cdots \times \beta_m \rightarrow S.\]

Definition 3.5. The \( s_{i+1} \) restriction of \( \Delta(s_i) \) is the function

\[
\Delta[s_{i+1}] : \beta_{i,1} \times \cdots \times \beta_{i,n_i} \rightarrow S
\]  

(37)

defined as follows:

\[
\Delta[s_{i+1}](s_i)(\phi_{i,1}, \ldots, \phi_{i,n_i}) = \\
\text{if } C_{i+1}(\phi_{i,1}, \ldots, \phi_{i,n_i}) \rightarrow s_{i+1} \\
\text{fi}
\]

Note that \( \Delta[s_{i+1}](s_i)(\phi_{i,1}, \ldots, \phi_{i,n_i}) = s_{i+1} \) whenever \( \Delta(s_i)(\phi_{1}, \ldots, \phi_{m}) = s_{i+1} \), and that
\( \Delta[s_{i+1}](s_i)(\phi_{i,1}, \ldots, \phi_{i,n_i}) = \bot \) whenever \( \Delta(s_i)(\phi_{1}, \ldots, \phi_{m}) \not= s_{i+1} \).

Definition 3.6. Let \( [T_m^1]^{\phi_{m,1}^1} \) be a virtual data stream, and \( r(q) = [s_0, \ldots, s_r] \) a run at time \( t \) for the query \( q \). The stream is weakly safe for \( r(q) \) if, for every \( t+k \) such that \( [T_h]^{\phi_{h,t}}(t+k) = \bot \) there is a \( s_{k+1} \) restriction of \( \Delta \) that does not depend on \( \phi_h \).

Weakly safe streams have the same relevant properties as safe streams. In particular, the following holds.

Theorem 3.2. Let the virtual stream \( [T_m^1]^{\phi_{m,1}^1} \) be weakly safe for the run \( r(q) = [s_0, \ldots, s_r] \) at time \( t \). Then \( r(q) \) accepts \( [T_m^1]^{\phi_{m,1}^1} \) if and only if it accepts \( \phi_{m,1}^1 \).

Proof. Consider the transition \( s_i \rightarrow s_{i+1} \) of the run. We have to show that the query does this transition on the virtual stream if and only if it does it on the original stream. The state transition function \( \Delta(s_i) \) can be rewritten in terms of its \( s \)-restrictions as

\[
\Delta(s_i)(\phi_1, \ldots, \phi_m) = \\
\text{if } \Delta(s_0)(\phi_{0,1}, \ldots, \phi_{0,n_0}) \not= \bot \rightarrow \Delta[s_0](\phi_{0,1}, \ldots, \phi_{0,n_0}) \\
\text{fi}
\]

\[
\Delta(s_1)(\phi_{1,1}, \ldots, \phi_{1,n_1}) \not= \bot \rightarrow \Delta[s_1](\phi_{1,1}, \ldots, \phi_{1,n_1})
\]

\[
\vdots
\]

\[
\Delta(s_{i+1})(\phi_{i+1,1}, \ldots, \phi_{i+1,n_{i+1}}) \not= \bot \rightarrow \Delta[s_{i+1}](\phi_{i+1,1}, \ldots, \phi_{i+1,n_{i+1}})
\]

\[
\vdots
\]

\[
\Delta(s_r)(\phi_{r,1}, \ldots, \phi_{r,n_r}) \not= \bot \rightarrow \Delta[s_r](\phi_{r,1}, \ldots, \phi_{r,n_r})
\]

\[
\text{fi}
\]
The transition that actually takes place is \( s_i \rightarrow s_{i+1} \), so, only the function
\[
\Delta(s_{i+1})(\phi_{i+1,1}, \ldots, \phi_{i+1,n_{i+1}})
\]
evaluates to something different from \( \bot \). At time \( t+k \), then, we can replace the transition function with the function
\[
\Delta(s_{i+1})(\phi_{i+1,1}, \ldots, \phi_{i+1,n_{i+1}}).
\]
This function depends only on \( \phi_{i+1,1}, \ldots, \phi_{i+1,n_{i+1}} \) so, by the previous theorem, if we set all other features to \( \bot \), the result of the run won’t change.

If the state \( s_i \) occurs again in the run, with a transition \( s_i \rightarrow s_k \), we can rewrite the query marking the first state as \( s_1 \) the first time it occurs, \( s_2 \) the second time, and so on. We will then apply a different substitution every time.

4 Algorithms

The previous considerations suggest that, at every time, we should compute only the features that are needed at that time. The problem, of course, is that in general one doesn’t know which transition will take place for a query until the query actually makes the transition. One can, however, start computing all the features and block the computation of a feature as soon as one realizes that it is not needed at a given step.

In this section I shall consider a system with a set of features \( \Phi \), which are computed on a stream, and a set of queries \( Q \) which are being executed concurrently on the stream. Each \( q \in Q \) is a finite state machine. \( S[q] \) is the set of states of query \( q \), and, for \( u \in S[q] \), \( \Phi[q,u] \subseteq \Phi \) is the set of features necessary to compute the state transition function \( \Delta_q(u) \) of query \( q \) at state \( u \), which depends on the conditions \( C^q[u], \ldots, C^q_k[u] \); \( s[q] \) is the current state of the query \( q \) at that time. For each feature there is a set \( \Theta[\phi] \), which contains the queries that are waiting for the completion of the computation of \( \phi \).

**Example:** Consider the queries \( q_1, q_2, \) and \( q_3 \) of the previous example, and assume (without loss of generality for the present purposes) that all the features are computed in a single step. Suppose that the three queries are executed, and that, before reading the next element from the stream, \( q_1 \) is in state \( s_1 \), \( q_2 \) is also in state \( s_1 \), and \( q_3 \) is in state \( s \). The two features \( \phi \) and \( \psi \) will then have the following sets of waiting queries:

\[
\phi : \{q_1, q_2, q_3\} \\
\psi : \{q_1, q_2\}
\]

Upon the arrival of the data element \( x(n) \), the following operations should take place:

i) for each feature \( \phi \), set \( \Theta[\phi] = \emptyset \);

ii) each query \( q \)--which, at time \( n \) is in state \( s[q] \)--adds itself to the set of all features it uses: \( \forall \phi \in \Phi[q,s[q]] \rightarrow \Theta[\phi] \leftarrow \Theta[\phi] \cup \{q\} \). At the end of this step the following property is true:

\[
\forall q \forall \phi \phi \in \Phi[q,s[q]] \iff q \in \Theta[\phi] 
\tag{38}
\]

iii) each feature \( \phi \) with \( \Theta[\phi] \neq \emptyset \) begins computation of \( \phi \);
iv) each query starts the computation of the conditions necessary for its transition functions, and each condition waits for the completion of the features it needs;

v) as soon as a condition is satisfied for query $q$, $q$ removes itself from the set $\Theta$ of all the features in its feature set, that is, it executes

$$\forall \phi \in \Phi[q, s[q]]\hspace{1em} \Theta[\phi] \leftarrow \Theta[\phi] - \{q\}.$$  

vi) each feature $\phi$ left with $\Theta[\phi] = \emptyset$ stops execution and sets $\phi_x(n) = \perp$.

These operations are implemented by the algorithms of figure 1. The first loop of

$$\text{process}(s:\Sigma(\alpha))$$

while true do

foreach $\phi \in \Phi$ do

$\Theta[\phi] \leftarrow \emptyset$

od

$(d, s) \leftarrow s();$

foreach $q \in Q$ do

foreach $\phi \in \Phi[q, s[q]]$ do

$\Theta[\phi] \leftarrow \Theta[\phi] \cup \{q\};$

od

od

foreach $\phi \in \Phi$ do

if $\Theta[\phi] \neq \emptyset$ then

mutex($\phi$);

ftcalc($\phi, d$);

fi

od

foreach $q \in Q$ do

trans($q, s[q]$)

if comp($C_1, q, s[q]) \rightarrow s[q] := s_1; remove(q)$;

fi

od

foreach $\phi \in \Phi[q, s[q]]$ do

$\Theta(\phi) \leftarrow \Theta(\phi) - \{q\};$

if $\Theta(\phi) = \emptyset$ then

kill(comp($\phi$));

fi

od

Figure 1: Algorithms for implementing the virtual stream procedure in the case in which the features are instantaneous.

The function $\text{process}$ determines which features will have queries waiting for them. The second initializes a semaphore for each feature and starts computation, and the third initiates the transition function for each query. The function $\text{ftcalc}$ will compute the corresponding feature in input $d$ and then signal its completion on the corresponding semaphore:
Algorithm to compute feature $\phi$ with data $d$

Then signal its completion on its semaphore.

The function $trans$ is basically an implementation of the transition function of a query. The only twists are that the conditions are computed by the function $comp$, which waits for the features to compute, and that before the state transition the function $remove$ is called, which removes the query $q$ from the sets of those waiting for feature completion, stops the computation of all the conditions of $q$ that are still waiting to complete and stops feature computations as required by step iv).

The running time of $comp$ and $remove$ is $O(Q + n)$, where $Q$ is the number of queries and $n$ the number of features used by each query; the running time of $process$ is $O(nQ \log^* nQ)$ (using the appropriate disjoint data set structure [10]), that is, to all practical purposes, $O(nQ)$. It must be pointed out that all these execution times are given in terms of elementary operations and that, in any practical setting, the execution time of all the algorithms is a fraction of the execution time of a single feature computation.

The important properties of this algorithm are summed up in the following two theorems:

**Theorem 4.1.** If all the feature computation algorithms terminate, each execution of the while loop in the function $process$ executes without being blocked by any of the semaphores.

(The function $process$ itself, of course, never terminates execution on an infinite stream.) The proof of this theorem is fairly straightforward given the form of the algorithms, and is not reported here.

**Theorem 4.2.** The algorithm generates a weakly safe stream for each query $q \in Q$.

**Proof.** Suppose the theorem is not true. Then there must be a query $q$ and a transition $s_i \rightarrow s_{i+1}$ of a run for the query that is not executed. Writing the transition function $\Delta(s_i)$ as in the proof of theorem 3.2, this means that the function $\Delta(s_{i+1})(s_i)$ is not executed. None of the other conditions is true, so $\Delta(s_{i+1})(s_i)$ doesn't give true only if one of the features $\phi_1, \ldots, \phi_{i,n_i}$ is set to $\perp$ during the computation of the step. Let $\bar{\phi}$ be the feature for which this is the case.

In the first part of the function loop, the query $q$ is placed in the set $\Theta(\bar{\phi})$, so that in the second part, the set $\Theta(\bar{\phi})$ is non-empty, and the function $comp(\bar{\phi})$ begins execution. Therefore, the only way in which $\bar{\phi}$ obtains a value $\perp$ is through a call to $kill(comp(\bar{\phi}))$ in some of the functions $remove$. But in $remove$, the function $comp(\bar{\phi})$ can only be stopped if $\Theta(\bar{\phi})$ is empty. Since at the beginning $q \in \Theta(\bar{\phi})$ before we have $\Theta[\phi] = \emptyset$ the operation $\Theta[\phi] \leftarrow \Theta[\phi] - \{q\}$ must have been executed. This operation is executed in $remove(q)$, which is executed only after that the transition $s_i \rightarrow s_{i+1}$ takes place, which contradicts our hypothesis. $\square$
Note that the results of the algorithm are not safe in the stricter sense of definition 3.4. Consider for example the simple query

\[ C_1(\phi_1) \rightarrow S_1 \]
\[ C_2(\phi_2) \rightarrow S_2 \]

which has a transition function from \( s_0 \)

\[ \Delta(s_0) : \beta_1 \times \beta_2 \rightarrow S. \] (40)

In this query we have \( \Phi[q,s_0] = \{\phi_1,\phi_2\} \) so if, say, \( C_1 \) ends first, then the algorithm sets \( \phi_2(t) = \bot \) but \( \Delta(s_0) \) also depends on \( \phi_2 \), so the virtual stream is not safe in the sense of definition 3.4.

5 Features with long computation

The algorithm in the previous section assumed that the features were strongly causal, that is, that every feature depended only on one element of the feature stream. For many features this is not the case: even a simple moving average can be computed only based on a certain number of past samples.

In this section, I shall relax the assumptions of the previous algorithms to allow computations extended in the number of samples. In order to simplify the notation, I will assume that all the features are computed over \( m \) samples. A feature \( \phi \) is computed through \( m \) calls to the function \( \text{comp} \). If \( s \) is the stream, over which the feature \( \phi \) is computed, then a value of \( \phi \) can be obtained by:

\[
\text{for } k=0 \text{ to } m-1 \text{ do}
\]
\[
(v, s) \leftarrow s();
\]
\[
\text{ftcalc}(\phi, v, k);
\]
\[
\text{od}
\]

that is, the function \( \text{ftcalc} \) takes as parameters the feature that is being computed, the current value extracted from the stream, and the index of the current computation step. In practice, the function \( \text{compute} \) will have to maintain a state of the computation, but this is not a concern in the present situation.

If the number of samples on which the computation is based varies, I shall assume that \( m \) is the largest of them; if a feature requires \( m' < m \) steps, I shall formally replace it with a feature that requires \( m \) steps and for which the first \( m - m' \) are dummies that perform no computation.

Since in this situation a feature begins computation before it is actually needed, it will be necessary to implement some sort of "look ahead" mechanism. Consider a feature \( \phi \); every time a sample arrives, say sample \( t_0 \), we will begin a new computation of the feature \( \phi \), just on the off-chance that some query will need it.

If nothing comes to stop the computation, it will be ready at time \( t_0 + m - 1 \), when it will be used by a query to make a state transition. So, every query must keep track, at time \( t_0 \), of the features it might need at time \( t_0 + 1, t_0 + 2, \ldots, t_0 + m - 1 \). If, because of some transition, it determines that it no longer needs feature \( \phi \) at some time in the feature, say \( t_0 + k \), it will signal so. If no query needs feature \( \phi \) at
time \( t_0 + k \), then the computation that was calculating the feature \( \phi \) and that was
going to end at time \( t_0 + k \) can be stopped.

Figure 2 shows the situation schematically for a feature \( \phi \) with \( m = 5 \). Let us call

\[ k \phi \]

the computation of \( \phi \) that will finish computing \( k \) samples from the present. In
figure 2 at time \( t_0 \) there are 5 ongoing computations of the feature \( \phi \). The
computation \( 0 \phi \) has started at time \( t_0 - 4 \) and at \( t_0 \) is doing its final computation
step: unless this computation is stopped (because every query does a transition
without having to wait for it), the current value of \( \phi \), that is, \( \phi(t_0) \), will be
available at the end of this computation, and will be used for a query transition.
The three features whose computation has begun at times \( t_0 - 3 \), \( t_0 - 2 \), and \( t_0 - 1 \) (that
is, \( 1 \phi \), \( 2 \phi \), and \( 3 \phi \)) are in the process of being computed. If the situation evolves in
such a way that no query will possibly need the feature \( \phi \) at time \( t_0 + k \) then the

\[
\text{corresponding computation can be stopped. Once a computation has been stopped, it}
can no longer be resumed, since the stream values on which it would be based have
already passed and disappeared from the system. Finally, the computation of \( 4 \phi \) will
begin at this time if some query determines that it might possibly need it at time

\( t_0 + 4 \). In order to transform these qualitative considerations into a provably
correct algorithm, I shall begin by formalizing the expression \( \text{query } q \text{ might need}
feature } \phi \text{ } k \text{ samples from now.}

As before, given a feature \( \phi \), the symbol \( k \phi \) will refer to the instance of \( \phi \) that will
be ready \( k \) steps from the present time. Let \( Q \) be a set of such feature instances.
The anticipation of \( Q \) is defined as

\[
\downarrow Q = \{ k^{-1}_\phi \mid k \phi \in Q \land k \geq 1 \} \tag{41}
\]

while the postponement of \( Q \) is defined as

\[
\uparrow Q = \{ k^{+1}_\phi \mid k \phi \in Q \land k \leq m - 1 \}. \tag{42}
\]

Given a state \( s \) for a query \( q = (S, \Delta, s_0, F) \), the set of possible followers of \( s \) is

\[
\chi(s) = \{ s' \in S : s' \in \text{Im(\Delta(s))} \} \tag{43}
\]

that is, \( \chi(s) \) is the set of states of \( q \) that can be reached from \( s \) in a single
transition. In the common graph representation of a state machine, \( \chi(s) \) would
correspond to the set of neighbors of \( s \). The \( k \) step future prediction of a state \( s \)
is defined as recursively as follows:
\[
\Phi[q,s](0) = \{^0\phi : \phi \in \Phi[q,s]\} \\
\Phi[q,s](k) = \bigcup_{s' \in \chi(s)} \Phi[q,s'](k-1)
\]

where, in the first equation, \(\Phi[q,s]\) is the set of features needed by \(q\) in order to compute the transition function from state \(s\), as defined in the previous section. This set enjoys some important properties.

Theorem 5.1. Let \(^u\phi \in \Phi[q,s](k)\), then \(u = k\).

Proof. The proof is by induction on \(k\). For \(k = 0\) the theorem is trivially true by construction. Assume now that it is true for \(k\), and let \(^u\phi \in \Phi[q,s](k+1)\). Then, by definition, there is \(s'\) such that \(^u\phi \in \Phi[q,s'](k)\), that is, by definition of \(\uparrow\), that \(u = v + 1\) with \(^v\phi \in \Phi[q,s'](k)\). By the inductive hypothesis, it is \(v = k\), that is, \(u = k + 1\).

In other words, \(\Phi[q,s](k)\) is a set of feature instances whose computation will finish \(k\) time steps from the present.

Theorem 5.2. Given a query \(q\), in a state \(s\), \(k\phi \in \Phi[q,s](k)\) if and only if there is a sequence of states \([s_0, \ldots, s_k]\) with \(s_0 = s\), \(s_i \in \chi(s_{i-1})\) for \(i = 1, \ldots, k\), and \(\phi \in \Phi[q,s_k]\).

Proof. Suppose \(k\phi \in \Phi[q,s](k)\). I shall prove the existence of the path by induction. For \(k = 0\), if \(^0\phi \in \Phi[q,s](0)\), then \(\phi \in \Phi[q,s]\) by construction. Suppose now that the theorem is true for \(k - 1\), and let \(^k\phi \in \Phi[q,s](k)\). Then, by construction, there is \(s_1 \in \chi(s_0)\) such that \(^k\phi \in \uparrow \Phi[q,s_1](k-1)\), that is, such that \(^{k-1}\phi \in \Phi[q,s_1](k-1)\). By the inductive hypothesis, there is a path \([s_1, \ldots, s_k]\) such that \(s_i \in \chi(s_{i-1})\) and \(\phi \in \Phi[q,s_k]\). Since \(s_1 \in \chi(s_0)\), the path that we are looking for is \([s_0, s_1, \ldots, s_k]\).

Suppose now that a path \([s_0, s_1, \ldots, s_k]\) with the required characteristics exists. If \(k = 0\) the theorem is true by construction, so we will assume \(k > 0\) and prove it by contradiction.

Assume that it is false. Since for \(k = 0\) it is true, let \(p > 0\) be the smallest integer for which there is a path \([s_0, \ldots, s_p]\) for which the theorem is false, that is, the shortest path which satisfies the hypotheses of the theorem and for which there is \(\phi \in \Phi[q,s_p]\) and \(^p\phi \not\in \Phi[q,s_0](p)\).

Since \(^p\phi \not\in \Phi[q,s_0](p)\), there is no \(s' \in \chi(s)\) such that \(^p\phi \in \uparrow \Phi[q,s'](p-1)\). In particular, \(^p\phi \not\in \Phi[q,s_1](p-1)\), which implies that \(^{p-1}\phi \not\in \Phi[q,s_1](p-1)\). Then, for the path \([s_1, \ldots, s_p]\) (which, being a sub-path of \([s_0, \ldots, s_p]\), satisfies the hypotheses of the theorem) we have that \(^{p-1}\phi \not\in \Phi[q,s_1](p-1)\) and \(\phi \in \Phi[q,s_p]\). That is, the theorem is false for \([s_1, \ldots, s_p]\), which contradicts the hypothesis that \([s_0, \ldots, s_p]\) was the shortest path for which the theorem was false.

This theorem states that, given a query \(q\) in state \(s\), \(\Phi[q,s](k)\) contains all features that \(q\) might possibly need \(k\) steps from state \(s\), and only those. This is the set that the algorithm is going to use for prediction. If at time \(t_0\) a query \(q\) is in state \(s\), then the only features that \(q\) might possibly need at time \(t_0 + k\) are those contained in \(\Phi[q,s](k)\).
5.1 Query algorithms

In order to write down the query algorithms, I shall need to define a few more
structures.

i) $C[\phi,k]$, with $\phi \in \Phi$, $k = 0,\ldots,m - 1$ is the feature computation matrix. The element
$C[\phi,k]$ contains data about $\phi^k$, that is, it contains data about the computation of
$\phi$ that will finish $k$ steps from the present. These data consist of the
quantities:

a) $C[\phi,k].q$ is the set of queries that might need feature $\phi$ $k$ steps from the
present.

b) $C[\phi,k].p$ is the process that is computing the corresponding step of the
feature computation, that is, $C[\phi,k].p$ is executing $calc(\phi, v, m-k-1)$.
Whenever $C[\phi,k].q$ becomes empty, $C[\phi,k].p$ is stopped.

ii) $\zeta[\phi]$ is a temporary storage where we place the queries that require a new
computation of $\phi$ to begin at the next time step.

Example: Consider again the stream with the features $\phi$ (motion) and $\psi$ (dimensions),
and the query:

$q_4$: an object that moves in the first quadrant after it had an area greater than $\alpha$
for two consecutive frames.

The query is represented by the automaton:

\[
\begin{array}{c}
 s_0 \\
 \overset{Z}{\rightarrow} \\
 s_1 \\
 \overset{Z}{\rightarrow} \\
 s_2 \\
 \overset{Z \land W_1}{\rightarrow} \\
 s_3 \\
\end{array}
\]

(45)

Consider the automaton in state $s_0$. We have:

$\Phi[q_4,s_0](0) = \{0\psi\}$
$\Phi[q_4,s_0](1) = \{1\psi\}$
$\Phi[q_4,s_0](2) = \{2\phi^2\psi\}$

that is, at times 0 and 1, the query will under no circumstance need the feature $\phi$.
The matrix $C.q$ is

\[
\begin{array}{cccc}
 \phi, k & 0 & 1 & 2 \\
 \phi & 0 & 0 & \{q_4\} \\
 \psi & \{q_4\} & \{q_4\} & \{q_4\} \\
\end{array}
\]

(Note that, being $m = 2$, the matrix $C.q$ has only two columns; the third one was added
here for illustrative purposes.)

*   *   *   *
process(s: $\Sigma(\alpha)$)

while true do
  foreach $\phi \in \Phi$ do
    $\Theta[\phi] \leftarrow \emptyset$
  od
  $(v, s) \leftarrow s()$
  foreach $\phi \in \Phi$ do
    foreach $k \in \{0, \ldots, m-1\}$ do
      if $C[\phi, k].q \neq \emptyset$ then
        $C[\phi, k].p \leftarrow \text{exec(ftcalc}(\phi, v, m-k-1))$
      fi
    od
  od
  foreach $q \in Q$ do
    trans(q, s);
  od
  step(C)
  od

removeB(q, c)
  c.q $\leftarrow$ c.q $\setminus$ \{q\};
  if c.q = $\emptyset$ then
    kill(c.p);
  fi

step(c)
  foreach $\phi \in \Phi$ do
    for i=1 to m-2 do
      $C[\phi, i] \leftarrow C[\phi, i+1]$
    od
    $C[\phi, m-1] \leftarrow \zeta[\phi]$
    $\zeta[\phi] \leftarrow \emptyset$
  od

trans(q, s)
  if comp(C1, q, s) then $s' \leftarrow s1$;
  if comp(C2, q, s) then $s' \leftarrow s2$;
  :
  if comp(Cn, q, s) then $s' \leftarrow sn$;
  fi

removeA(q, s)
  foreach $k \in \{1, \ldots, m-1\}$ do
    $R \leftarrow \Phi[q,s](k) - \uparrow \Phi[q,s'](k-1)$
    foreach $h_{\phi} \in R$ do
      removeB(q, C[\phi, k]);
    od
  od
  foreach $m-1 \phi \in \Phi[q, s'](m-1)$ do
    $\zeta[\phi] \leftarrow \zeta[\phi] \cup \{q\}$
  od

removeA(q, s)
  foreach $c \in C[q]$ do
    kill(comp(c));
  od
  foreach $\phi \in \Phi[q, s]\{0\}$ do
    $C[\phi, 0] \leftarrow C[\phi, 0] \setminus \{q\}$
    if $C[\phi, 0].q = \emptyset$ then
      kill(C[\phi,0].p);
    fi
  od

Figure 3: Algorithms for implementing the virtual stream procedure in the case in which the features have long computation.
The general idea of the algorithm is quite similar to that of the previous section. The function \textit{process} does the general computation: for every new datum from the stream it starts the calculation of all the feature instances that have queries waiting, and it calls the appropriate state transition functions (see figure 3) Note that, in the previous example, only the feature \( \psi \) will start computing at time 0, since \( m = 2 \) (it takes two steps to compute the longest computing feature), and \( C.q[\phi,0] = C.q[\phi,1] = \emptyset \).

The function \textit{trans}(q) waits until enough features have been computed to evaluate one of the conditions of q, and then removes q from the waiting list of the features it doesn't need anymore. In this case, it will also have to consider the features that will be ready in future steps. This function also places the query q in the "waiting set" of those features that some path starting from the state \( s' \) might need \( m-1 \) steps after the transition, that is, of the features that this query require start computing in the next step. The functions \textit{removeA} and \textit{removeB} are quite similar to their counterpart of the previous section. The function \textit{step} shifts the contents of the matrices \( C \) of all the features, marking the passage of a time unit.

When all the queries have finished their transition and all the features have finished computing, the algorithm gets ready for the next step. At this point, the matrix element \( C[\phi,k] \) contains the computation that will be ready \( k \) steps from now, that is, has computed \( m-1-k \) of its \( m-1 \) computations. In the next step, the feature will compute its step \( m-k = m-1-(k-1) \), that is, it will have to be stored in the element \( C[\phi,k-1] \) of the computation matrix. The computation of \( C[\phi,m-1] \) has already ended at the current time step, and it can be removed from the system. The elements \( C[\phi,m-1] \) will contain the computations that will begin with the next sample of the stream, viz. those that have queries waiting in their set \( \zeta \).

\textbf{Example:} Consider again the previous situation, and assume that the query does the transition from state \( s_0 \) to state \( s_1 \). The call \textit{removeA} in the function \textit{trans} will not do anything in this case, since the only feature that was being computed was \( \psi \), whose computation must already have ended for the transition to take place.

The portion \((\gamma)\) is executed with \( k = 1 \) only (since, in this case, we have \( m = 2 \)), and looks for features that in state \( s_0 \) might have been necessary in the future but that in state \( s_1 \) aren't any longer. In this case

\[ \Phi[q_4,s_0](1) = \Phi[q_4,s_1](0) = \{ \psi \} \]  

so that \( R = \emptyset \) and the query \( q_4 \) is not removed from the waiting list of any feature. In the part \((\alpha)\) we look at the features whose computation should begin at this time by looking ahead at the features that, starting from state \( s_1 \), we might need \( m-1 \) steps from now. Here we have

\[ \Phi[q_4,s_1](1) = \{ \phi, \psi \} \]  

so we set \( \zeta[\phi] = \zeta[\psi] = \{ q_4 \} \). When executing the function \textit{step}, those values will be entered in the last column of the matrix \( C,q \) (in the portion \((\beta)\)), causing their execution to begin. At the new time, the matrix \( C,q \) is

\[
\begin{array}{ccc}
\phi & 0 & 1 \\
\phi & \emptyset & \{ q_4 \} \\
\psi & \{ q_4 \} & \{ q_4 \}
\end{array}
\]

There will be two instances of the feature \( \psi \) being computed: one that will finish computing at the current time, and the other that is being prepared for finishing at
Suppose now that, at the next transition, when $\psi$ terminates computing, the condition $Z$ turns out to be false. The query then goes back to $s_0$ and, in $(\gamma)$, we have

$$\Phi[q_4, s_1](1) = \{1_\phi, 1_\psi\}$$
$$\uparrow \Phi[q_4, s_0](0) = \{1_\psi\}$$

that is, $R = \{1_\phi\}$. In other words, the computation of $\phi$ that terminated at time $t_0 + 1$ and that might have been necessary in $s_1$ is no longer necessary. In $s_1$ it was necessary to begin computing $\phi$ because of the possibility that the query do the transition $s_1 \rightarrow s_2$, after which $\phi$ is necessary in order to decide whether the next transition was $s_2 \rightarrow s_3$ or $s_2 \rightarrow s_0$. But, once in $s_0$, there is no path whose condition will need $\phi$ in one time step, so it is useless to continue its computation. In the function removeB the query $q_4$ will be removed from the waiting list of the computation $\uparrow \phi$ and, being the list left empty, its computation will be aborted.

5.2 Correctness

The data structure $C$ is initiated by executing $k$ "dry runs", that is, computing the features, putting them in $\zeta$, and then calling the function step. The matrix $\Phi$ is given as part of the structural description of the queries, and one can assume that it is a by-product of the conversion of the query into a finite state machine. The time complexity of this algorithm is in this case $O(nQk)$.

In order to prove the properties of the algorithm---in particular that the algorithm computes all the features necessary for the query---I shall consider a run

$$[s_1, \ldots, s_t, \ldots, s_{t+m-1}]$$

(48)

that ends at time $t + m - 1$ with a state transition that involves features whose computation began at time $t$. I am assuming that the length of the run is greater than $m$. If this is not the case, since the queries are running continuously, one can build a run of the necessary length by concatenating several runs. Remember that a run is the sequence of states that a query goes through while responding to the inputs of the original (non-virtual) feature stream (that is, the stream in which no feature computation is stopped) and, since the queries are continuously running, one can draw from it sequences of states of any length one desires.

Consider the state $s_{t+m-1}$ in the run (48). Then the properties of the following two lemmas hold. The proof of both lemmas is in appendix.

Lemma 5.1. Let $\phi \in \Phi[q, s_{t+m-1}]$. Then upon arrival of the sample $d_t$ of the stream (that is, the one that was read while the query was in state $s_t$) the call compute($\phi$, $d_t$, 0) was executed.

In order to clarify the content of the second lemma, consider how the computation of a feature progresses and how it is tracked in the matrix $C$. At time $t$, the process that computes $\phi$ is started and inserted in $C[\phi, m-1], p$. At every step the computation of the feature proceeds one step further, and the process is moved to the previous column of the matrix. At time $t + m - 1$ the process arrives at the element $C[\phi, 0]$. The path followed by the process can be represented schematically in the
following diagram:

\[
\cdots \rightarrow s_t \rightarrow \cdots \rightarrow s_{t+(m-1)-p} \rightarrow \cdots \rightarrow s_{t+(m-1)-r} \rightarrow \cdots \rightarrow s_{t+(m-1)} \rightarrow \cdots
\]  

\[
C[\phi, m-1] \rightarrow C[\phi, p] \rightarrow C[\phi, r] \rightarrow C[\phi, 0]
\]

comp. start \hspace{1cm} comp. ends

where \(0 \leq r < p \leq m-1\). As usual, I will talk of a time step \(\tau\) referring to the part of the execution that processes the element \(d_\tau\) of the stream; at this time the query is in state \(s_\tau\). The proof is in appendix.

Lemma 5.2. Suppose that \(\phi \in \Phi[q, s_{t+(m-1)}]\) and that, at step \(t + (m - 1) - p\) one has \(C[\phi, p].q \neq \emptyset\). Then, for all \(r\) such that \(0 \leq r \leq p\), at step \(t + (m - 1) - r\) it is \(C[\phi, r].q \neq \emptyset\).

The algorithm stops the computation of a feature whenever \(C[\phi, u].q\) becomes empty. The following theorem guarantees that this will not result in the loss of any useful feature, that is, that once \(C[\phi, u].q\) becomes empty it will stay empty.

Theorem 5.3. Suppose that, at step \(t + (m - 1) - p\), it is \(C[\phi, p].q = \emptyset\). Then for all \(u\) with \(0 \leq u \leq p\) at time \(t + (m - 1) - p\) it will be \(C[\phi, u] = \emptyset\).

The proof is a trivial consequence of the fact that queries are only inserted in \(C[\phi, m-1].q\) by the function \(\text{step}\). From that moment on, the only operation possible on \(C[\phi, u]\) is the removal of features.

With these results we can move on to derive the proof of the correctness of the algorithm:

Theorem 5.4. The function \(\text{process}\) (and its sub-programs) generates a weakly safe stream for each query \(q \in Q\).

Proof. (sketch) Consider any transition \(s \rightarrow s'\) in a run on the query on the complete stream and assume, by contradiction, that with the function \(\text{process}\) this transition is not made.

Lemma 5.1 guarantees that all the features \(\phi \in \Phi[q, s]\) begin computing \(m - 1\) steps before this transition, and lemma 5.2 guarantees that their computation is never stopped. So, when the query is in state \(s\), all the features needed to compute the transition function \(\Delta[s]\) are entering their last step of computation.

The only way the features necessary for the transition \(s \rightarrow s'\) might fail to compute is if they are stopped because \(C[\phi, 0]\) becomes empty.

This situation is analogous to that of single step features, and theorem 4.2 proves that the transition \(s \rightarrow s'\) takes place.

6 How much do we save?

The main idea of virtual streams is that, by stopping the computation of a feature whenever it is not needed, one can save CPU cycles. One important question, of course, is how much one can save and how this alleged gain depends on the
characteristics of the system. In this section I shall propose some calculations to try and answer these questions under somewhat simplified assumptions. The main simplifications of the model of this section are:

i) each feature is instantaneous, that is, its value depends only on one element of the stream, and

ii) each branch of each query contains a condition on a single feature.

If a feature $φ$ is present in one of the $c$ conditions of query $q$, then I will say that $φ$ covers $q$. This means that, if $φ$ finishes computing while $q$ is still waiting on its conditions, $q$ will proceed to the next state. The computation ends when enough features have been computed to cover all the queries. In other words, one is interested in the probability that $k$ features will be sufficient to cover all the queries but, in order to solve a slightly general problem, I will determine the probability that, in a system with $F$ features and $Q$ queries, $f$ features ($f \leq F$) cover $q$ queries ($q \leq Q$).

The first step of this determination is to calculate the probability that a given feature will cover a given query. The probability that a given feature will appear in a given condition of the query is $1/F$, and the probability that a given feature will not appear in any of the $c$ conditions of the query is $(1 - 1/F)^c$. The probability that a given feature will appear in at least one of the $c$ conditions is then

$$\alpha = 1 - \left(1 - \frac{1}{F}\right)^c.$$  \hspace{1cm} (50)

The probability that a feature cover $k$ queries is given by all possible combinations in which the feature can be present in $k$ of the $Q$ queries and absent in the $Q - k$ remaining ones. This value is

$$\Xi(Q, F, c; k) = \binom{Q}{k} \alpha^k (1 - \alpha)^{Q - k}.$$  \hspace{1cm} (51)

Consider now $f$ features, and the probability that the first of them covers $k_1$ queries, the second cover $k_2$ more queries, and so on, until the $f$th covers $k_f$ as yet uncovered queries. The probability that the first feature (out of $F$) covers $k_1$ queries is $\Xi(Q, F, c; k_1)$. The second feature is now drawn out of $F - 1$ remaining features, and must cover $k_2$ of the remaining $Q - k_1$ queries, with a probability $\Xi(Q - k_1, F - 1, c; k_2)$. The $f$th feature is drawn out of $F - f + 1$, and must cover $k_f$ of the remaining queries, which are in number of $Q - \sum_{j=1}^{f-1} k_j$; the probability of this happening is

$$\Xi \left(Q - \sum_{j=1}^{f-1} k_j, F - f + 1, c; k_f\right).$$  \hspace{1cm} (52)

The probability of the whole combination is therefore

$$\prod_{i=1}^{f} \Xi \left(Q - \sum_{j=1}^{i-1} k_j, F - i + 1, c; k_i\right).$$  \hspace{1cm} (53)

The probability that $f$ features cover $q$ queries is the sum of all probabilities (53) over all combinations $k_1, \ldots, k_f$ such that $k_1 + \cdots + k_f = q$, that is,

$$P(Q, F, c, f, q) = \sum_{k_1 + \cdots + k_f = q} \prod_{i=1}^{f} \Xi \left(Q - \sum_{j=1}^{i-1} k_j, F - i + 1, c; k_i\right).$$  \hspace{1cm} (54)
The relation (54) is in excellent agreement with the results obtained through simulation. Figures 6 and 6 show the predicted fraction of the features that have to be computed to advance all the queries in the system for various values of the number of queries, of the number of features, and of the number of conditions in each query. One important consequence of (54) (although quite an obvious one) is that the fraction of features whose computation can be avoided depends only on the number of features, the number of queries, and the number of conditions on each state of the queries. It doesn’t depend on the nature of the feature or of the system. Methodologically, this is important because it allows us to dispense with "realistic" simulations that, depending on many uncontrollable variables, result in anecdotal evaluation of doubtful generalizability. In this case, we know from (54) that we only need to control three variables to determine the performance of the algorithms. Note that, for the purposes of simulation, the actual state of a query is irrelevant, since we assume that the conditions distribution is independent of the state. Therefore, each query was assigned a random subset of features, the size of which depended on the parameters of the test. A random feature was selected, all the queries that contained that feature were removed from the waiting set of all the features they contained, and all the features whose waiting set was empty were
counted as "stopped". The process was repeated until all features had either been selected (viz. counted as completed) or stopped. The fraction of the features that it is necessary to compute increases, as it is intuitively obvious, with the number of queries, and the largest percentages tend to appear when there are few features and many queries. As the number of queries tends to infinity, the fraction of features computed tends, of course, to 1, but as the number of queries increases, the growth in the fraction of features tends to flatten out so that adding queries to an already large pool doesn't degrade the performance too much. There is also a very strong trend towards the reduction of the fraction of features as $c$ increases. The explanation of this phenomenon is quite obvious: as the number of conditions in a query increases, the query is dependent on a larger number of features, and one of them is sufficient to make it move to another state. Because of this, each feature will in general cover more queries.

Note that a first glance it might appear that the fraction of completed features is a function of the ratio $f/q$, but analyzing equation (54) one can see that this is not the case. This is confirmed statistically with a simple $t$-test on the simulation results.

7 Conclusions

In this paper I have presented algorithms for querying what I have called virtual feature streams that is, informally speaking, feature streams "with holes": portions of the stream in which the features are not computed because they are provably unnecessary for the computation of the queries that are currently running in the system.

The paper has presented the algorithms and has proved that they are safe, that is, that no query that would run on the complete feature stream will cease to do so because the stream is virtual, and no query that would not accept the complete stream will accept the virtual one. There are indications that the algorithm might be optimal given the information available at any time, that is, that it doesn't compute any feature that is provably unnecessary at the time of computation. So far no theorem in this sense has been proved.

The paper has presented an evaluation of the fraction of features that one might expect to have to compute for the virtual stream, under the simplifying hypothesis that each branch of the query contains a condition on a simple features.

A more complete model, allowing the presence of several features in the same condition has not been developed yet. Intuitively, what it will do is to make a query with $c$ branches out of a state behave like one with $c' < c$ branches, but the extent of this effect hasn't been determined as yet.

Acknowledgments

This work was supported in part by Consejería de Educación de la Comunidad Autónoma de Madrid, under the grant CCG08-UAM/TIC/4303, Búsqueda basada en contexto como alternativa semántica al modelo ontológico and by the program Ramón y Cajal of the Ministerio de Educación y Ciencia.
A Proofs of the lemmas

Proof of lemma 2.1. The proof is by induction on \( k \). For \( k = 0 \) we have
\[
F[x](0) = (\pi_1 \circ r)()
\]
by definition.
Suppose now that the lemma is true for \( k \). Then we have
\[
F[x](k + 1) = (\pi_1 \circ \pi_2 \circ r)(k)
\]
(by definition of \( F \))
\[
= (\pi_1 \circ \pi_2 \circ \pi_2 \circ r)(() )
\]
(by the inductive hypothesis) (55)

\[\square\]

Proof of lemma 2.2. I shall prove by induction that, for each \( k \geq 0 \), we have
\[
F[G[q]](k) = q(k).
\]
For \( k = 0 \) we have
\[
F[G[q]](0) = \pi_1(G[q])
\]
(by definition of \( F \))
\[
= q(0)
\]
(by definition of \( G \)) (56)
Suppose now that the theorem is true for \( k \). Then
\[
F[G[q]](k + 1) = (\pi_1 \circ \pi_2^{k+1} \circ G[q])(() )
\]
(by lemma 2.1)
\[
= (\pi_1 \circ \pi_2 \circ (\pi_2 \circ G[q]))()
\]
(property of the function composition)
\[
= (\pi_1 \circ \pi_2 \circ G[\lambda u. q(u + 1)])(() )
\]
(by definition of \( G \))
\[
= F[G[\lambda u. q(u + 1)]](k)
\]
(by lemma 2.1)
\[
= (\lambda u. q(u + 1))(k)
\]
(by the inductive hypothesis)
\[
= q(k + 1)
\]
(57)

\[\square\]

Proof of lemma 5.1. There is a path of length \( m \) from \( s_t \) to \( s_{t+m-1} \) so, by theorem 5.2,
\[
m^{-1} \phi \in T[q, s_t](m - 1).
\]
The query was at time \( t \) in state \( s_t \) so in step \( \alpha \) (algorithm trans), \( q \) has been placed in \( \zeta[\phi] \). Therefore, after trans has been executed, \( \zeta[\phi] \neq \emptyset \) and, in statement \( \beta \) (algorithm step), \( C[\phi, m - 1, q] \neq \emptyset \), and, in the next iteration of the process the function \( \text{compute}(\phi, d_t, 0) \) will be executed.

\[\square\]

Proof of lemma 5.2. Suppose the lemma is false. Then there is a step \( t + (m - 1) - v \) at which \( C[\phi, v] \) becomes empty, with \( 0 \leq v \leq p \). For this to happen, there must be a step \( t + (m - 1) - u \) with \( 0 \leq v \leq u \leq p \) at which query \( q \) is removed from \( C[\phi, u, q] \).
At this time, feature $\phi$ is $u$ steps away from completion and, since $q$ is removed from the set of queries waiting for it (in step $\gamma$ of the function trans), it must be

$$u\phi \in T[q, s_{t+(m-1)-u}] | (u) - |T[q, s_{t+(m-1)-(u-1)}] | (u-1)$$

(58)

Since this is the time step in which the query is removed, until this step the query needed feature $\phi$, so

$$u\phi \in T[q, s_{t+(m-1)-u}] | (u)$$

(59)

therefore (58) implies

$$u\phi \not\in T[q, s_{t+(m-1)-(u-1)}] | (u-1)$$

(60)

that is

$$u^{-1}\phi \not\in T[q, s_{t+(m-1)-(u-1)}] | (u-1)$$

(61)

there is a run of length $u-1$ from $s_{t+(m-1)-(u-1)}$ to $s_{t+(m-1)}$ and, by hypothesis $\phi \in \Phi[q, s_{t+(m-1)}]$, we conclude from theorem (5.2) that

$$u^{-1}\phi \in T[q, s_{t+(m-1)-(u-1)}] | (u-1)$$

(62)

which is a contradiction.

References


Path sharing and predicate evaluation for high-performance xml filtering. AMC

steerable flow fields. In Proceedings of the IEEE Computer Society Conference on


[14] Todd J. Green, Gerome Miklau, Makoto Onizuka, and Dan Suciu. Processing xml
streams with deterministic automata. In Proceedings of the 9th international
conference on database theory (ICDT), 2003.

Gordon Anderson. Sase: Complex event processing over streams. In Proceedings of


[17] Rune Hjelsvold and Roger Midtstraum. Modelling and querying video data. In
Proceedings of teh 20th VLDB Conference. Modelling and querying video data. In

the 21st International Conference on Data Engineering, 2005.

system. In Proceedings of the International Conference on Database Engineering,
2005.

international conference on pattern recognition, 1990.


[22] Utkarsh Srivastava, Kamesh Munagala, and Jennifer Widom. Operator placement for
in-network stream query processing. In Proceedings of the twenty-fourth ACM

[23] W. Thomas. Languages, automata, and logic. In Grzegorz Rozenberg and Arto
Salomaa, editors, Handbook of formal languages. Heidelberg:Springer-Verlag,
1997.

initial delays. In Proceedings of the 29th international conference of very

video indexing. In Proceedings of the IEEE international symposium on circuits
and systems, 1997.