Simultaneous Equation Systems for Query Processing on Continuous-Time Data Streams

Yanif Ahmad, Olga Papaemmanouil, Uğur Çetintemel, Jennie Rogers
Brown University
{yna, olga, ugur, jennie}@cs.brown.edu

Abstract— **We introduce Pulse, a framework for processing continuous queries over models of continuous-time data, which can compactly and accurately represent many real-world activities and processes. Pulse implements several query operators, including filters, aggregates and joins, that work by solving simultaneous equation systems, which in many cases is significantly cheaper than processing a stream of tuples. As such, Pulse translates regular queries to work on continuous-time inputs, to reduce computational overhead and latency while meeting user-specified error bounds on query results. For error bound checking, Pulse uses an approximate query inversion technique that ensures the solver executes infrequently and only in the presence of errors, or no previously known results.**

We first discuss the high-level design of Pulse, which we fully implemented in a stream processing system. We then characterise Pulse's behavior through experiments with real data, including financial data from the New York Stock Exchange, and spatial data from the Automatic Identification System for tracking naval vessels. Our results verify that Pulse is practical and demonstrates significant performance gains for a variety of workload and query types.

I. Introduction

Many physical processes and activities exhibit limited and predictable state changes over time. The temperature of a specific region, wind gust values as observed by a weather station, and the trajectory of cars on a highway segment can all be modeled using algebraic, continuous models of time. While the underlying processes being monitored are fundamentally continuous, they are sampled to produce a stream of discrete values, which are then fed to stream processing engines as input. Our work is motivated by the key observation that practical, continuous models are not only capable of accurately and compactly representing many such discrete data streams, but, in many cases, can also be processed (using relational operators) much more efficiently than the discrete input values they represent.

In this paper, we present Pulse, a framework to study the viability of continuous-time models as inputs to continuous query processing. Pulse transforms a continuous query operating on discrete input data streams during planning, to instead operate on continuous-time models. Our work focuses on designing a query processing plan to operate on these models — Pulse uses continuous-time models as a compact representation of the input data stream and processes this compact representation to reduce computation overhead and consequently end-to-end latency. We consider polynomials as models whose structure, when combined with a query's structure, allows us to represent discrete queries as simultaneous equation systems. Continually solving this system allows us to determine when the discrete-time query produces results and consequently the values of output tuples. For example consider a query detecting collisions of moving objects, expressed as a join comparing the objects' proximity:

```
select from objects R
  join objects S on (R.id <> S.id)
  where abs(distance(R.x, R.y, S.x, S.y)) < c
```

While standard stream processors compare many position samples, Pulse is able to analytically solve models of object trajectories to determine query results. Pulse includes continuous-time implementations of a variety of relational-style stream operators, including filters, aggregates, and joins.

Pulse further facilitates a tradeoff between the result error tolerance and query efficiency. For each query, users supply an error bound for the results. Pulse inverts these error bounds specified at query outputs to bounds at query inputs. Error bound inversion poses several interesting challenges including how to handle non-invertible multi-input operators such as joins and aggregates. Pulse addresses these challenges by maintaining the lineage of query execution, and by using heuristics to apportion bounds across inputs to optimize validation efficiency.

Pulse provides two operating modes from an application's viewpoint. The first is an online predictive processing scenario, where Pulse uses predictive models of unseen data and precomputes query results off into the future. The second is offline historical processing for scenarios such as a large number of "what-if" queries over historical data streams. Here, Pulse computes a model of the historical stream once, and feeds it as input to the transformed queries, significantly reducing their execution times. These operating modes begin to highlight the joint space of application-level functionality and system optimizations that we envision exploring with Pulse.

In summary, our contributions are:

1) We present a holistic design for a stream processor capable of operating on continuous-time models of input data and handling any errors in these models.
2) We describe a novel implementation of a query plan as a simultaneous equation system per operator.
3) Our queries are capable of guaranteeing error bounds on query results efficiently by inverting results and bounds to be validated at query inputs.

4) We have fully prototyped our design in an actual stream processing engine, and evaluated its performance against realistic workloads from two canonical continuous-time stream applications.

This paper is laid out as follows. Section II presents an overview of Pulse, including its operating two modes, predictive and historical processing, and its data streams and query models. In Section III we present the predictive transformation and its application to filters, joins and aggregates. Section IV presents our strategy for performing validation to ensure desired accuracy bounds are met. Section V presents an experimental evaluation using an implementation of Pulse in the Borealis [1] stream processing engine prototype. Finally we conclude in Sections VI and VII with discussion of relevant work and future directions on this topic.

II. Pulse Framework Overview

Continuous-time models provide two distinctive properties for use in query processing: they facilitate random access to data at arbitrary points in time, and enable a compact representation of the data as model parameters. In this section we present an overview of the nature of the application types, data streams and queries we support.

A. System Model

In this section, we discuss two novel uses of models in the query execution model of a stream processing engine in terms of both functionality and performance.

Predictive Processing. In the predictive processing scenario, Pulse uses its modeling component to generate the continuous-time models for unseen data values off into the future, processes these predicted inputs, and generates predicted query results, all before the real input data becomes available from external sources. This style of predictive processing has important uses both from the end-application perspective (e.g., a traffic monitoring application can predict congestions at on road segments and send alerts to drivers) and system optimization perspective (e.g., predictive results can mask I/O latencies, or network latencies in wide-area network games by pre-fetching).

Historical Processing. The second scenario is off-line historical data analysis that involves running a large number of "parameter sweeping" or "what-if" queries (common in the financial services domain). Applications replay a historical stream as input to a large number of queries with different user-supplied analytical functions or a range of parameter values. The results are then typically compared against each other and what was obtained in the past, to identify the "best" strategy or parameters to use in the future. In historical processing, Pulse's modeling component is used to generate a continuous-time model of the historical stream that can be stored and used as an input to all historical queries. Thus, the cost of modeling can be amortized across many queries.

Query: `SELECT * from A MODEL` $A.x = A.x + A.vt$
`JOIN B MODEL` $B.y = B.vt + B.at^2$
`ON(A.x < B.y)`

Transformation	Description
$A.x < B.y$	
$A.x - B.y < 0$	difference equation
$A.x + A.vt - (B.vt + B.at^2) < 0$	substitute models
$A.x + (A.v - B.v)t - B.at^2 < 0$	factor time variable t

Fig. 1. Pulse transforms predicates in selective operators to determine a system of equations whose solution yields the time range containing the query result.

B. Data Stream Model

Pulse adopts the following assumptions on the uniqueness and temporal properties of data stream attributes.

Modeled Attributes. For predictive processing, Pulse supports declarative model specification as part of its queries via a `MODEL`-clause, as shown in Figure 1. Query developers provide *symbolic* models defining a modeled stream attribute in terms of other attributes on the same stream and a variable t. For example in Figure 1, stream A has a modeled attribute $A.x$ defined in terms of *coefficient* attributes $A.x$ and $A.v$. We allow the self-reference to attribute $A.x$ since we build *numerical* models from actual input tuples where the values of all coefficient attributes are known. In this example, the model $A.x = A.x + A.vt$ represents the x-coordinate a moving object as its position varies over time from some initial position. We consider time-invariant piecewise polynomial models since they are often used for simple and efficient approximation. The symbolic form of a general nth degree polynomial for a modeled attribute a is: $a(t) = \sum_{i=0}^{n} c_{a,i} t^i$. To ensure a closed operator set, we restrict the class of polynomials supported to those with non-negative exponents, since it has been shown that semi-algebraic sets are not closed in the constraint database literature [14]. In historical processing our modeling component computes coefficient attribute values internally.

Temporal attributes. We assume each input stream S includes two temporal attributes, a reference attribute denoting a monotonically increasing timestamp globally synchronized across all data sources, and a delta attribute T. Pulse uses the reference timestamp's monotonicity to bound query state and delta timestamps for simplified query processing. Our models are piecewise functions, that is they are made up of *segments*. Denoting r as a reference timestamp and t^l, t^u as offsets, a segment, $s \in S$, is a time range $[r + t^l, r + t^u)$, for which a particular set of coefficients for a modeled attribute, $\{c_i\}$, are valid (written as $s = ([t^l_i, t^u_i), c_i) = ([t^l, t^u), c)_i$). In the remainder of this work, we drop the reference timestamp r from our time ranges for readability. We adopt the following update semantics. For two adjacent input segments overlapping temporally, the successor segment acts as an update to the preceding segment for the overlap, that is $\forall i, j : [t^l, t^u)_i \cap [t^l, t^u)_j \neq \emptyset \wedge [t^l, t^u)_i < [t^l, t^u)_j \Rightarrow (([t^l_i, t^l_j), c_i), \ldots, ([t^l, t^u), c)_j)$. This captures the uniqueness properties of an online piecewise function, where pieces appear sequentially.

Key attributes. Pulse's data streams contain exactly two other types of attributes, keys and unmodeled attributes. Keys are discrete, unique attributes and may be used to represent discrete entities, for example different entities in a data stream of moving object locations. Unmodeled attributes are constant for the duration of a segment, as required by our time-invariant models. We omit details on the operational semantics of the core processing operators with respect to key and unmodeled processing due to space constraints. Our general strategy is to process these using standard techniques alongside the modeled attributes.

III. Continuous-Time Processing

The underlying principle of our continuous-time processing mechanism is to take advantage of the temporal continuity provided by the input streams' data models in determining the result of a query. The basic computation element in Pulse is a simultaneous equation system that is capable of performing computation on continuous functions corresponding to operations performed by the relational algebra. In this section we describe how we construct these equation systems from our data models for core operators such as filters, aggregates and joins, and how we are able to compose these equation systems to perform query processing.

A. Selective Operator Transform

Selective operators, such as stream filters and joins, produce outputs upon the satisfaction of a predicate comparing input attributes using one of the standard relational operators (i.e., $<, \leq, =, != , \geq, >$). We derive our equation system by transforming predicates in a three step process. Consider the a predicate with a comparison operator R, relating two attribute x, y as xRy. Our transformation is:

	General form
1. Rewrite in difference form	x - y R 0
2. Substitute continuous model	x(t) - y(t) R 0
3. Factorize model coefficients	(x-y)(t) R 0

We provide an example of these steps as applied to a join operator in Figure 1. The above equation defines a new function, $(x-y)(t)$, from the difference of polynomial coefficients that may be used to determine predicate satisfaction and consequently the production of results. Note that we are able to simplify the difference form into a single function by treating the terms of our polynomials independently. Depending on the operator R and the degree of the polynomial, there are various efficient methods to approach the above equation. In the case of the equality operator, standard root finding techniques, such as Newton's method or Brent's method [3], solve for points at which $(x-y)(t) = 0$. We may combine root finding with sign tests to yield a set of time ranges during which the predicate holds. We illustrate this geometrically in Figure 2.

The above difference equation forms one row of our equation system. By considering more complex conjunctive predicates, we arrive at a set of difference equations of the

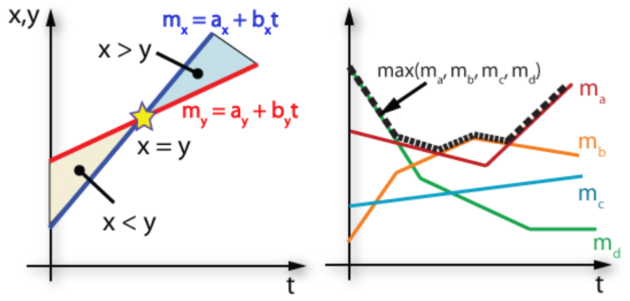

Fig. 2. A geometric interpretation of the continuous transform, illustrating predicate relationships between models for selective operators, and piecewise composition of individual models representing the continuous internal state of a `max` aggregate.

above form that must all hold simultaneously for our selective operator to produce a result. That is, given the following predicate and models: $x_1 R_1 y_1 \wedge x_2 R_2 y_2 \wedge \ldots \wedge x_p R_p y_p$, where $\forall i. x_i = \sum_{j=0}^{d} c_{x,i}^{j} t^i$, and $\forall i. y_i = \sum_{j=0}^{d} c_{y,i}^{j} t^i$, and $c_{x,i}^{j}$ is the jth coefficient in a segment, we derive the following equation system:

$$\begin{bmatrix} c_{x,1}^0 - c_{y,1}^0 & \cdots & c_{x,1}^d - c_{y,1}^d \\ c_{x,2}^0 - c_{y,2}^0 & \ddots & \vdots \\ \vdots & & \\ c_{x,p}^0 - c_{y,p}^0 & \cdots & c_{x,p}^d - c_{y,p}^d \end{bmatrix} \mathbf{t} \begin{bmatrix} R_1 \\ R_2 \\ \vdots \\ R_p \end{bmatrix} \mathbf{0}$$
$$= \mathbf{Dt}\ \mathbf{R}\ \mathbf{0} \qquad (1)$$

In the above equation, $\mathbf{t}$ represents a vector comprised of powers of our time variable (i.e., $[t, t^2, t^3, \ldots]'$). Thus the above equation system has a single unknown variable, namely a point in time t. We denote the matrix $\mathbf{D}$ our difference equation coefficient matrix. Under certain simplified cases, for example when $\mathbf{R}$ consists solely of equality predicates (as would be the case in a natural or equi-join), we may apply efficient numerical algorithms to solve the above system (such as Gaussian elimination or a singular value decomposition). A general algorithm involves solving each equation independently and determining a common solution based on intersection of time ranges. In the case of general predicates, for example including disjunctions, we apply the structure of the boolean operators to the solution time ranges to determine if the predicate holds. Clearly, Equation 1 may not have any solutions indicating that the predicate never holds within the segments' time ranges for the given models. Consequently the operator does not produce any outputs.

Pulse uses update segments to drive the execution of our equation systems. Consider the arrival of a segment, with time range $[t_0, t_1)$. For a filter operator, we instantiate and solve the equation system from the contents of the segment alone, ensuring that the solution for the variable t is contained within $[t_0, t_1)$ (for both point and range solutions). For a join, we use equi-join semantics along the time dimension, specifically we execute the linear system for each segment $[t_2, t_3)$ held in state that overlaps with $[t_0, t_1)$ (for each attribute used in the

Operator	Inputs	State	Implementation	Outputs
Filter	x_i	–	$\mathbf{D} = [x_i - c_i]$; **solve** $\mathbf{DtR0}$	$\{(t, x_i) \vert \mathbf{DtR0}\}$
Join	x_i on left input y_i on right input	order-based segment buffers, $S_x = \{([t^l, t^u), s_x) \vert t^l > t_y\}$ $S_y = \{([t^l, t^u), s_y) \vert t^l > t_x\}$	align x_i, y_i w.r.t t; $\mathbf{D} = [x_i - y_i]$; **solve** $\mathbf{DtR0}$	$\{(t, x_i, y_i) \vert \mathbf{DtR0}\}$
Aggregate **min, max**	x_i	state model, $S = \{([t^l, t^u), s) \vert t^l > t_x - w\}$	align x_i, s_i w.r.t t $\mathbf{D} = [x_i - s_i]$; **solve** $\mathbf{DtR0}$	$\{(t, s_i) \vert \mathbf{DtR0}\}$
Aggregate **sum, avg**	x_i	segment final $C = \int_{t^l}^{t^u} \sum_{i=0}^{d} x_i t^i$, $wf_{tail} = \int_{t-w}^{t} \sum_{i=0}^{d} x_i t^i dt$	$wf_{sum} =$ $wf_{tail} + C + wf_{head}$	$([t_l, t_u], wf_{sum})$
Aggregate group-by, function f	x_i per group	state for f per group	hash-based group-by, impl for f per group	outputs for f per group

Fig. 3. Operator transformation summary. Symbol definitions: x_i, y_i are polynomial coefficients for attributes x, y; $t = [t^l, t^u)$ is the valid time range for a segment; t_x, t_y denote the reference timestamps for the latest valid times for attributes x, y; (t, x) is the segment itself as a pair of valid times and coefficients; $(t, s_x)_i$ is a segment of attribute x that is kept in an operator's state; s_i are the coefficients of these state segments.

predicate). In our solver, we only consider solutions contained in $[t_0, t_1) \cap [t_2, t_3)$.

B. Aggregate Operator Transform

Aggregation operators have a widely varying set of properties in terms of their effects on continuous functions. In this section we present a continuous-time processing strategy for commonly found aggregates, namely min, max, sum, and average. At a high-level, we handle min and max aggregates by constructing an equation system to solve when to update our aggregate's internal state, while for sum and average, we define continuous functions for computing the aggregate over windows with arbitrary endpoints (i.e., continuous windows).

Min, max aggregates. The case of a single model per stream is trivial for min and max aggregates as it requires computing derivatives on polynomial segments to determine state updates. We focus on the scenario where a data stream consists of multiple models due to the presence of key attributes. The critical modification for these aggregates lies in the right-hand side of the difference equation, where we now compare an input segment to the partial state maintained within the operator from aggregating over previous input segments. We denote this state as $s(t)$, and define it as a sequence of segments: $s(t) = (([t^l, t^u), c)_1, ([t^l, t^u), c)_2 ..., ([t^l, t^u), c)_n)$, where each $([t^l, t^u), c)_i$ is a model segment defined over a time range with coefficients c_i. For example with a min (or max) function, the partially aggregated model $s(t)$ forms a lower (or upper) envelope of the model functions as illustrated in Figure 2. Thus we may write our substituted difference form as $x(t) - s(t)\ R\ 0$. This difference equation captures whether the input segment updates the aggregated model within the segment's lifespan. We use this difference equation to build an equation system in the same manner as for selective operators.

Sum, average aggregates. The sum aggregate has a well-defined continuous form, namely the integration operator. However, we must explicitly handle the aggregate's windowing behavior especially since sum and average aggregate along the temporal dimension. To this end, we define *window functions*, which are functions parameterized over a window's closing timestamp to return the value produced by that window. At a high level, window functions help to preserve continuity downstream from the aggregate. We now describe how we compute a window function for sums.

We assume a window of size w and endpoint t, and consider two possible relationships between this window and the input segments. The lifespan of a segment $[t^l, t^u)$ may either match (or be larger than) the window size w, or be smaller than w. In the first case, we may compute our window results from a single segment. Specifically, we claim that a segment covering $[t^l, t^u)$ may produce results for a segment spanning $[t^l + w, t^u)$, since windows closed in this range are entirely covered by the segment. We define the window function for this scenario as:

$$wf_{sum}(t) = \int_{t-w}^{t} \sum_{i=0}^{d} c_i t^i dt = \sum_{i=i}^{d+1} \frac{c_{i-1}}{i} t^i \qquad (2)$$

which is parameterized by the closing timestamp t of the window. In the scenario where a window spans multiple segments, we divide the window computation into three sub-cases: i) segments $[t_1^l, t_1^u)$ entirely covered by the window, ii) segments $[t_2^l, t_2^u)$ overlapping with head of the window t, and iii) segments $[t_3^l, t_3^u)$ overlapping with the tail of the window $t - w$. In the first sub-case, we compute the integral value for the segment's lifespan and denote this the constant C. In the second sub-case, we use the window function defined in Equation 2, and refer to this as the *head integral*. For the third sub-case, we apply an integral spanning the common time range of the segment $[t_3^l, t_3^u)$, and window: $\int_{t-w}^{t_3^l} \sum_{i=0}^{d} c_i t^i dt$. We refer to this integral as the *tail integral*. Note that for a given segment t_3^l is known and fixed. However we are still left with the term $t - w$ in our formula, but can leverage the window specification which provides a fixed value of w to express the result of the integral, by expanding terms of the form $(t - w)^i$ for $i > 0$ by the binomial theorem. This yields the following window function for windows spanning multiple segments: $wf_{sum}(t) = \int_{t-w}^{t_3^l} \sum_{i=0}^{d} c_i t^i dt + C + \int_{t_2^l}^{t} \sum_{i=0}^{d} c_i t^i dt$

For every input segment $[t_i^l, t_i^u)$ at the aggregate, we compute and cache the segment integral C, in addition to a function for the tail integral. This metadata is to be used by windows functions produced by future updates arriving at the aggregate. Finally we produce a window function for the input segment itself that spans all windows contained in its time range by fetching segment integrals and tail integrals for the set of windows $[t^l - w, t^u - w)$. While the above discussion concerned a sum function, these results may easily be applied to compute window functions for averages as $wf_{avg} = \frac{wf_{sum}}{w}$.

Transformation Limitations. Frequency-based aggregates are those that fundamentally depend on the number of tuples in the input stream. Examples include count, frequency moments, histograms etc. Certain aggregation functions can be viewed as mixed aggregates if they depend on both the content and the frequency, for example a sum aggregate may have larger values for high rate data streams (assuming positive numbers). Presently, our framework does not handle frequency oriented aggregates, and can only handle mixed aggregates when their computation involves all tuples in the relation (and thus all points on the continuous function) like sum and average. Figure 3 summarizes Pulse's selective and aggregate operator transforms.

C. Query Transform

Pulse performs operator-by-operator transformation of regular stream query instantiating an internal query plan comprised of simultaneous equation systems. Each equation system is closed, that is it consumes segments and produces segments, enabling Pulse's query processing to use segments as a first-class datatype. However, depending on the operator's characteristics, an equation system may produce an output segment whose temporal validity is a single point. This occurs primarily with selective operators involving at least one equality comparison. The reduction of a model to a single point limits the flow of models through our representation, since the remaining downstream operators can only perform discrete processing on this intermediate result.

Once the processed segment reaches an output stream, we produce output tuples via a sampling process. For selective operators, this requires a user-defined sampling rate. We note that for an aggregate operator producing query results, there is no explicit need for a application-specified output rate. This may be inferred from the aggregate's window specification, and in particular the slide parameter which indicates the periodicity with which a window closes, and thus the aggregate's output rate.

IV. Validating Query Processing

To handle differences between our continuous-time models and the input tuples, Pulse supports the specification of accuracy bounds to provide users with a quantitative notion of the error present in any query result. We consider an absolute error metric and *validate* that continuous-time query results lie within a given range of results produced by a standard stream query. One validation mechanism could process input tuples with both continuous-time and regular stream queries and check the results. However, this naive approach performs duplicate computation, offsetting any benefits from processing inputs in a continuous form.

Our validation mechanism checks accuracy at the query's inputs and completely eliminates the need for executing the discrete-time query. We name this technique query inversion since it involves translating a range of output values into a range of input values by approximately inverting the computation performed by each query operator. Some operators that are many-to-one mappings, such as joins and aggregates have no unique inverse when applied to outputs alone. However we may invert these operators given both the outputs and the inputs that caused them, and rely on continuity properties of these inputs to invert the output range. Query inversion maintains these inputs as query lineage, compactly as model segments.

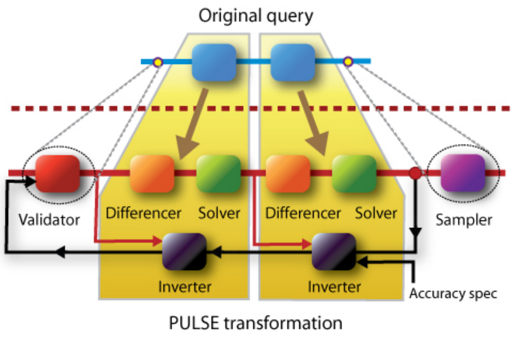

Fig. 4. High level overview of Pulse's internal dataflow. Segments are either given as inputs to the system or determined internally, and processed as first-class elements.

We use accuracy validation to drive Pulse's online predictive processing. In this scenario, Pulse only processes queries following the detection of an error. We note that accuracies may only be attributed to query results if the query actually produces a result. Given the existence of selective operators, an input tuple may yield a null result, leaving our accuracy validation in an undefined state. To account for this case, we introduce *slack* as a continuous measure of the query's proximity to producing a result. We define slack as:

$$\begin{aligned}&\textbf{slack} = \min_t \|\mathbf{Dt}\|_\infty\\ &\text{s.t} \quad t \in \bigcap [t^l, t^u)_i \qquad \forall i.[t^l, t^u)_{update} \cap [t^l, t^u)_i \neq \emptyset\end{aligned}$$

Above, we state that we only compute slack within valid time ranges common with the update segment causing the null (for stateful operators). Using the maximum norm ensures that we do not miss any mispredicted tuples that could actually produce results. Following a null any intermediate operator, Pulse performs slack validation, ignoring inputs until they exceed the slack range. Thus Pulse alternates between performing accuracy and slack validation based on whether previous inputs caused query results. Figure 4 provides a high-level illustration of Pulse's internal dataflow, including the inverter component that maintains lineage from each operation and participates in both accuracy and slack bound inversion.

A. Query Output Semantics

We briefly discuss the semantics of the outputs produced by continuous-time data processing. While a complete discussion of the topic lies outside the scope of this paper, we make several observations in the context of comparing and understanding the operational semantics of a continuous-time processor in comparison to a discrete-time processor. Clearly, the two modes of processing are not necessarily operationally equivalent on a given set of inputs. They may differ in the following ways.

Observation 1: Pulse may produce false positives with respect to tuple-based processing. If Pulse's query results are not discretized in the same manner as the input streams, Pulse may produce results that are not present under regular processing of the input tuples. For example, consider an equi-join that is processed in continuous form by finding the intersection point of two models. Unless we witness an input tuple at the point of the intersection, Pulse will yield an output while the standard stream processor may not, resulting in a superset output semantics.

Observation 2: Pulse may produce false negatives with respect to tuple-based processing. False negatives occur when the discrete-time query produces results but Pulsedoes not, yielding a subset output semantics. This may occur as a result of precision bounds which allow any tuple lying near its modelled value to be dropped. Any outputs that may otherwise have been caused by the valid tuple are not necessary, and therefore omitted. Again the difference in result sets arises from a lack of characterizing discretization properties.

B. Query Inversion

We describe query inversion as a two-stage problem, first as a local problem for a single operator, and then for the whole query, leveraging the solution to the first problem operator to produce an inversion data flow.

Bound inversion problem: *given an output value and a range at an operator, what range of input values produces these output values?* This problem may have many satisfying input ranges when aggregates and joins are present in the query. For example, consider a sum aggregate, and the range $[5, 10]$ as the output values. There are infinitely many multisets of values that sum to 5, (e.g. the sets $\{4, 1\}$ and $\{-1, -2, 8\}$). The fundamental problem here is that we need to identify a unique inverse corresponding to the actual computation that occurs (motivated by continuity for future bound validation). We use the following two properties to perform this restriction:

Property 1: continuous-time operators produce temporal sub-ranges as results. This ensures that every output segment is caused by a unique set of input segments.

Property 2: modelled attributes are functional dependents of keys throughout the dataflow. Each operator in our transformation preserves a functional dependency between keys and segments by passing along the key values that uniquely identify a segment.

These properties ensure we are able to identify the set of input segments for operations involving multiple segments (joins and aggregates) through segments' time ranges and key values. Providing we maintain the input keys and segments used to produce an intermediate operator's results (i.e., the lineage of a segment), we are able to identify the cause of each output segment, making query inversion an issue of accessing lineage. We remark that the cost of maintaining lineage is less prohibitive than with regular tuples due a segment's compactness (a full analysis of the lineage requirements lies outside this paper's scope).

Given both the input models and the output models, solving the bound inversion problem then becomes an issue of how to apportion the bound amongst the set of input models. We describe *split heuristics* in the next section to tackle this problem. Our run-time solution to the bound inversion problem is dynamic and expressive, providing the ability to adapt to changing data distributions. By considering both the input and output segments during inversion, we are able to support different types of bounds including both absolute and relative offset bounds.

Query inversion problem: *given a range of values on each attribute at a query's output, what ranges of query input values produce these outputs?* Query inversion determines the appropriate context for performing bound inversion, given the query's structure. In particular we focus on addressing attribute aliasing, and attribute dependencies caused by predicates, as shown in the following example. Consider the query (omitting windows and precision bounds for readability):

```
select a, b as x, d from R join S
    where R.a = S.a and R.a < S.d
```

Here, a new attribute x is declared in the results' schema, and is an alias of the attribute b. We must track this data dependency to support query inversion on error bounds specified on attribute x, and refer to this metadata as bound *translations*. The second type of dependency concerns query where-clauses. In this example, the attribute $S.d$ is not part of the query's results, but constrains the results via its presence in a predicate. We track these dependencies and refer to them as *inferences*. During the inversion process, we apportion bounds to attributes such as $S.d$, inferring the values they may take.

C. Accuracy and Slack Bound Splitting

In this section, we present two heuristics for allocating value ranges of an operator's output attributes to its input attributes for both accuracy and slack bounds. Pulse supports the specification of user-defined split heuristics by exposing the a function interface for the user to implement, for an absolute error metric. We describe our heuristics in terms of the function signature (simplified to a single modelled attribute a for ease of understanding):

$$\{(ik_p, [i^l_a, i^u_a]), ..., (ik_q, [i^l_a, i^u_a])\} = \\ split(ok, oc, [o^l, o^u], \{(ik_p, ic_a) \ldots, (ik_q, ic_a)\})$$

where $(ik_p, [i^l_a, i^u_a])$ are the bounds allocated to input attribute a for key p. Also, ok, oc denote the keys and coefficients of the output segment, $[o^l, o^u]$ the output bound, and finally $\{(ik_p, ic_a) \ldots (ik_q, ic_a)\}$ the keys and coefficients of the input segments producing the output. Note that the result of our split function includes both the set of input keys that we split over, in addition to the bounds. Thus bounds are only allocated to the keys that actually cause the output.

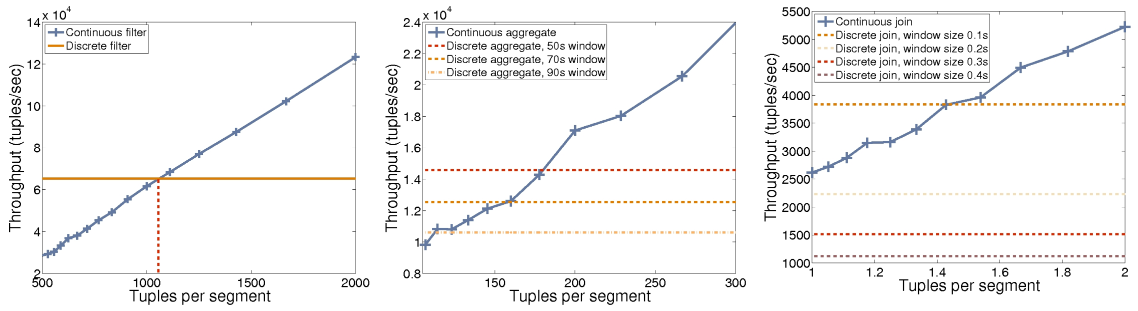

Fig. 5. Microbenchmarks for i) filter ii) aggregate and iii) join operators, all with a 1% error threshold.

Equi-split: this heuristic assigns the output error bound uniformly across all input attributes. Specifically, it implements the following split heuristic:

$$(ik_p, [i^l_a,\, i^u_a]) = [\tfrac{o^l}{n}, \tfrac{o^u}{n}]$$
$$\text{where} \quad a \in D(o), n = |\{ik_p \ldots ik_q\}| * |D(o)|,$$
$$D(o) = translations(o) \cup inferences(o)$$

The above equation specifies the uniform allocation of a bound to each key and attribute dependency.

Gradient split: this heuristic attempts to capture the contribution of each particular input model to the output result. Formally, the heuristic computes:

$$(ik_p, [i^l_a, i^u_a]) = \tfrac{d(ic_a)}{dt} * [\tfrac{o^l}{\sum_{m\in I} ic_m}, \tfrac{o^u}{\sum_{m\in I} ic_m}]$$
$$\text{where} \quad a \in D(o)$$
$$D(o) = translations(o) \cup inferences(o)$$
$$I = \{(ik_p, ic_a), \ldots, (ik_q, ic_a)\}$$

The above equation specifies that each bound allocated is the product of the gradient of a single segment with respect to the global segment of all input keys contributing to the result.

Both of the above schemes are conservative in the sense that they preserve two-sided error bounds, and ensure that the error ranges allocated on input attributes do not exceed the error range of the output attribute. A more aggressive allocation scheme may reduce two-sided error bounds to a one-sided error, for example in the case of inequality predicates, thereby improving the longevity of the bounds. In general, the efficiency of validating query processing is fundamentally an optimization problem and our current solution lays the framework for further investigation.

V. Experimental Evaluation

We implemented Pulse as a component of the Borealis [1] stream processing engine. This implementation provides full support of the basic stream processing operators including filters, maps, joins and aggregates and extends our stream processor's query language with accuracy and sampling specifications. Pulse is implemented in 27,000 lines of C++ code and adds general functionality for rule-based query transformations to Borealis, in addition to specialized transformations to our equation systems. We note that Pulse requires a small footprint of 40 lines in the core stream processor code base indicating ease of use other stream processors. In these experiments, Pulse executes on an AMD Athlon 3000+, with 2GB RAM,

Experiment	Parameter	Value
All	Page pool	1.5Gb
Filter	stream rate	6000-20000 tuples/sec
Aggregate	stream rate	20000-40000 tuples/sec
Join	stream rate	1000-10000 tuples/sec
Fig. 5i,ii,iii	precision bound	1%
Aggregate	stream rate	3000 tuples/sec
Fig. 7i	window	size 10-100s, slide 2s
	precision bound	1%
Join	stream rate	100-900 tuples/sec
Fig. 7ii	window	size 0.1s
	precision bound	1%
Historical	stream rate	3000-30000 tuples/sec
Fig. 8	window	size 60s, slide 2s
NYSE	stream replay rates	3000-8500 tuples/sec
Fig. 9i	precision bound	1%
AIS	stream replay rates	200-6000 tuples/sec
Fig. 9ii	precision bound	0.05%
Precision	stream rate	3000 tuples/sec
Fig. 9iii	precision bound	0.1-20%

Fig. 6. List of experimental parameters. Refer to MACD and "following" queries in Section VB for operator window sizes.

running Linux 2.6.17. We configured our stream processor to use 1.5GB RAM as the page pool for allocating tuples.

Our experiments use both a real-world dataset and a synthetic workload generator. The synthetic workload generator simulates a moving object, exposing controls to vary stream rates, attribute values' rates of change, and parameters relating to model fitting. Our real-world datasets are traces of stock trade prices from the New York Stock Exchange (NYSE) [6], and the latitudes and longitudes of naval vessels captured by the Coast Guard through the Automatic Identification System (AIS) [7].

A. Synthetic Workload Benchmarks

Our first results are a set of microbenchmarks for individual filters, joins and aggregates. We investigate the processing throughput for fixed size workloads from our moving object workload generator, under a varying model expressiveness measured as the number of tuples that fit a single model segment. The workload generator provides two-dimensional position tuples with a schema: x, y, v_x, v_y denoting x- and y-coordinates in addition to x- and y-velocity components.

Filter. Figure 5i demonstrates that the continuous-time implementation of a filter requires a strong fit in terms of the number of tuples per segment, between the model and the input stream. The continuous-time operator becomes viable at approximately 1050 data points per segment. This matches our intuition that the iterations performed by the linear system during solving dwarfs that performed per tuple by an extremely simple filter operation.

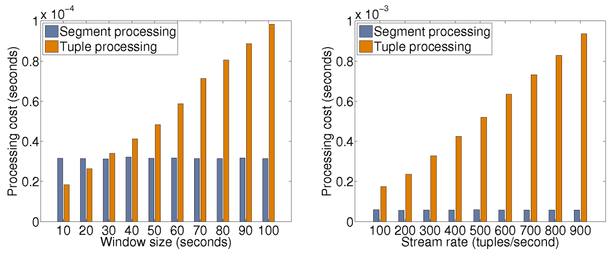

Fig. 7. Continuous-time and discrete processing overhead comparison for: i) aggregate operator, ii) join operator.

Aggregate. Figure 5ii compares the continuous-time aggregate's throughput for the min function under varying model fit settings. We also illustrate the cost of tuple-based processing at three window sizes for comparison. The window size indicates the number of open windows at any point in time, and thus the number of state increments applied to each tuple. This benchmark shows the continuous-time aggregate provides higher throughput at approximately 120-180 tuples per segment for different windows. Thus we can see that the model may be far less expressive (by a factor of 5x) for our processing strategy to be effective. This improvement primarily arises due to the increased complexity of operations per tuple performed by an aggregate, in comparison to linear system solving. Figure 7i illustrates the operator's processing costs as window sizes vary from 10 to 100 seconds. Here the cost of a tuple-based aggregate is clearly linear in terms of the window size, while the cost of our segment-based processing remains low due to the fact we are only validating the majority of tuples, and not solving the linear system for each tuple. We demonstrate that Pulse outperforms tuple processing at window sizes beyond 30 seconds, and is able to achieve a 40% cost compared to regular processing at a 100 second window.

Join. Figure 5iii displays the throughput achieved by a continuous-time join compared to a nested loops sliding window join as the number of tuples per segment is varied. The join predicate compares the x and y positions of objects in our synthetic workload. Figure 5iii shows that our join implementation outperforms the discrete join at 1.45 tuples per segment for a window size of 0.1s. This occurs because a nested loops join has quadratic complexity in the number of comparisons it performs, as opposed to the complexity of a validation operation which is linear in the number of model coefficients. Figure 7ii illustrates the difference in processing cost under varying stream rates, and clearly shows Pulse's significantly lower overhead. The processing cost of our mechanism remains low while the tuple-based cost increases quadratically (despite the linear appearance, we verified this

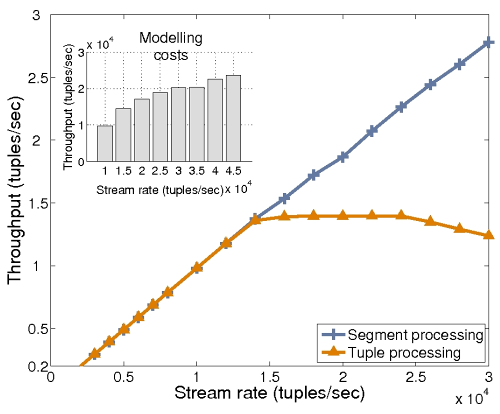

Fig. 8. Historical aggregate processing throughput comparison with a 1% error threshold.

in preliminary experiments while extending to higher stream rates). We plan on investigating this result with other join implementations, such as a hash join or indexed join, but believe the result will still hold due to the low overhead of validation compared to the join predicate evaluation.

Historical processing. Figure 8 presents throughput and processing cost results from the historical application scenario. In these results, we present the cost of performing model fitting, via an online segmentation-based algorithm [13] to find a piecewise linear model to the input data, in addition to processing the resulting segments. We consider a min aggregate, with a 60 second window, and a 2 second slide. Tuple processing reaches a maximum throughput of 15,000 tuples per second before tailing off due to congestion in the system as processing reaches capacity. In contrast, segment processing continues to scale beyond this point, demonstrating that the data modeling operation does not act as a bottleneck with this workload. The nested plot of modeling throughput, which executes our model fitting operator alone, illustrates that this instead happens at a higher throughput of approximately 40,000 tuples. This result indicates that data fitting is indeed a viable option in certain cases, and that simplistic modeling techniques such as piecewise linear models are indeed able to support high-throughput stream processing.

B. *NYSE and AIS Workloads*

We extracted the NYSE dataset of stock trade prices from the TAQ3 data release for January 2006, creating workloads of various sizes for replay from disk into Pulse. The schema of this dataset includes fields for *time, stock symbol, trade price, trade quantity*. In our experiments on this dataset, we stream the price feed through a continuously executing moving average convergence/divergence (MACD) query, a common query in financial trading applications. The MACD query is as follows (in StreamSQL syntax):

```
select symbol, S.ap - L.ap as diff from
  (select symbol, avg(price) as ap from
    stream S[size 10 advance 2]) as S
join
  (select symbol, avg(price) as ap from
    stream S[size 60 advance 2]) as L
on (S.Symbol = L.Symbol)
where S.ap > L.ap
```

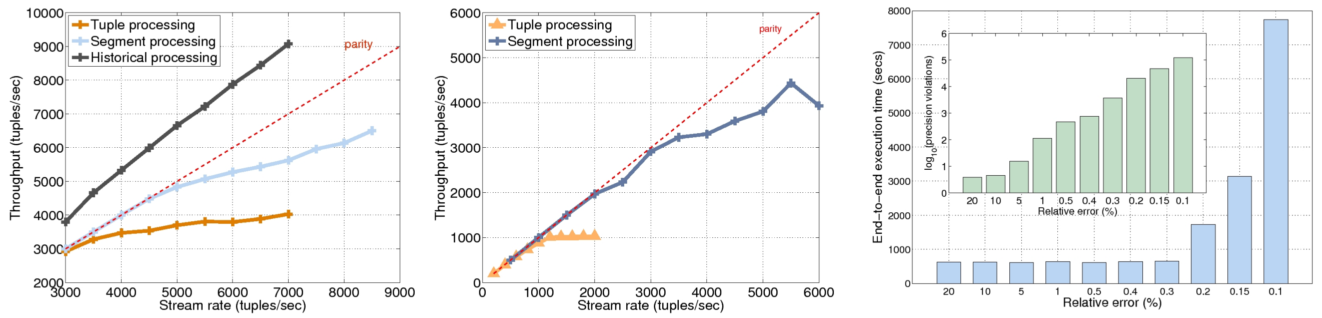

Fig. 9. NYSE and AIS dataset evaluation: i) Continuous-time processing of the NYSE dataset, with a 1% error threshold. ii) Continuous-time processing of the AIS dataset, with a 0.05% error iii) Continuous-time processing of the NYSE data at 3000 tuples/second.

This query uses two aggregate operations, one with a short window to compute a short-term average, and the other with a long window to compute a long-term average, before applying a join operation to check for the presence of a larger short-term average.

The AIS dataset contains geographic locations and bearings of naval vessels around the coasts of the lower 48 states over a 6-day period in March 2006, totalling to approximately 6GB of data. We extracted a subset of the data for replay, with the following schema: *vessel id, time, longitude, longitudinal velocity, latitude, latitudinal velocity*. We then use the following query to determine if two vessels were following each other:

```
select Candidates.id1, Candidates.id2, avg(dist)
  (select S1.id as id1, S2.id as id2,
     sqrt(pow(S1.x-S2.x,2) + pow(S1.y-S2.y,2)) as dist
   from S[size 10 advance 1] as S1
     join S as S2[size 10 advance 1]
     on (S1.id <> S2.id))[size 600 advance 10]
  as Candidates
group by id1, id2 having avg(dist) < 1000
```

The above query continuously tracks the proximity of two vessels with a join operation and computes the average separation over a long window. We then apply a filter to detect when the long-term separation falls below a threshold.

C. NYSE and AIS Processing Evaluation

In this section we compare the throughput of the NYSE and AIS datasets and queries as they are replayed from file.

Figure 9i compares the throughput Pulse achieves while processing the NYSE dataset in comparison to standard stream processing. In this experiment, we set error thresholds to 1% of the trade's value. We see that the tuple-based MACD query tails off at a throughput of approximately 4000 tuples per second. We ran no further experiments beyond this point as the system is no longer stable with our dataset exhausting the system's memory as queues grow. In contrast the continuous-time processor is able to scale to approximately 6500 tuples per second, and similarly begins to lead to instabilities beyond this point. As a further comparison, we plot the historical processing performance that represents the throughput of processing segments alone (without modelling). This reflects performance achieved following an offline segmentation of the dataset. Historical processing scales well in this range of stream rates due to the lack of any validation overhead. Also note that we achieve greater throughput than parity, due to lower end-to-end execution times for our fixed workload, through the early production of results from sampling the linear models.

Figure 9ii compares throughputs in the AIS dataset, for an error threshold of 0.05%. This plot illustrates that the original stream query tails off after a stream rate of 1100 tuples per second, achieving a maximum throughput of approximately 1000 tuples per second. In contrast, Pulse is able to achieve a factor of approximately 4x greater throughput with a maximum of 4400 tuples per second. We note the lower stream rate in the AIS scenario in comparison to the NYSE scenario, due to the presence of a join operator as the initial operator in the query (the MACD query has aggregates as initial operators). The segment processing technique reaches its maximum throughput without any tail off since it hits a hard limit by exhausting the memory available to our stream processor while enqueueing tuples into the system.

D. NYSE Performance vs. Precision Tradeoff

Figure 9iii displays the end-to-end processing latency achieved by Pulse for the MACD query on the NYSE dataset under varying relative precision bounds. The inset figure displays the number of precision bound violations that occurred during execution on a logarithmic scale. This figure demonstrates that Pulse is able to sustain low processing latencies under tight precision requirements, up to a threshold of 0.3% relative error for this dataset. The inset graph shows that as the precision bound decreases, there are exponentially more precision violations. Beyond a 0.3% precision, the processing latency increases exponentially with lower errors due to the queueing that occurs upon reaching processing capacity.

In summary, our experimental results show that continuous-time processing with segments can indeed provide significant advantages over standard stream processing providing the application is able to tolerate a low degree of error in the results. In certain scenarios Pulse is capable of providing up to 50% throughput gain in an actual stream processing engine prototype, emphasizing the practicality of the our proposed mechanisms.

VI. Related Work

To the best of our knowledge, Pulse is the first framework to process continuous queries on piecewise polynomials with simultaneous equation systems. Temporal query languages, stream [4], [18] and processing have all introduced temporal constructs such as windows and patterns to provide query functionality in the temporal domain but do not address continuity. Stream filtering mechanisms [19], [11] apply delta processing techniques to perform stateful query processing by establishing a bound (or predictive model) between a data source and a data processor. This allows the source to avoid sending updates, providing these updates adhere to the predictive model. These techniques are similar to the functionality provided by our accuracy splitter component, but do not consider processing issues for whole queries or computing with data models.

Neugebauer [20], and Lin and Risch [15] present interpolation techniques for base relations in relational databases. Deshpande and Madden [5] present a view-based approach to interpolated data points through the regression techniques, and advocate the use of standard query processing techniques upon the modeled data. In contrast, we attempt to maintain a model-based data representation throughout the query processing pipeline. Time series databases primarily focus on basic operations on a time-series datatype, and consider datatype-specific queries such as similarity search and subsequence matching [8], in addition to mining and analysis related queries [21]. Time-series modeling techniques include various segmentation algorithms used to capture time series as piecewise linear models [22], [13]. This represents a disjoint set of functionality from relational algebra and does not address the issue of how to leverage temporal continuity in high-volume relational data stream processing.

Moving object databases frequently leverage continuous-time models in the form of object trajectories during spatio-temporal query processing. Here, query types commonly include range search, range aggregation, spatial join [16] and nearest neighbour queries [23], and are often implemented through a specialized index structure for each query type. A survey of these indexes and access methods may be found in [17]. All of these works are related to Pulse, but do not capture the same query generality provided by Pulse's representation of queries as a composition of equation systems.

Finally, constraint databases have provided inspiration for representing high-volume data streams in a compact way, and for our formulation of continuous queries as linear systems. The DEDALE [9] project describes the implementation of a constraint database system that operates on infinite sets of tuples, simplifying tuples via normalization before processing via a constraint engine. Other works in the field have presented the constraint data and query model [12], and applied this to process queries over interpolated data [10].

VII. Conclusions and Future Work

This paper has described the Pulse framework, a data stream processor that works directly with continuous-time models to answer continuous queries posed by end users. We presented the basic workings of Pulse across a variety of relational operators including filters, joins and aggregates, and their composition as queries. We have developed a fully-functioning prototype in the Borealis stream processor, and used this to demonstrate that Pulse is able to yield significantly lower processing overheads in comparison to regular stream processing operators on real-world data and queries. For future work [2], we envisage segment indexing techniques to process highly segmented datasets resulting from many unmodeled attributes, potentially from a traditional relational database. We also plan to apply our techniques to other model types, including differential equations, time series, and frequency models such as Fourier series.

Acknowledgements. This work has been supported in part by the NSF under the grants IIS-0448284 and IIS-0325838. We thank Stan Zdonik and John Jannotti for their comments.

References

[1] D. Abadi et. al., "The design of the Borealis stream processing engine," in *CIDR*, 2005.
[2] Y. Ahmad and U. Çetintemel, "Declarative temporal models for sensor-driven query processing." in *DMSN*, 2007.
[3] R. P. Brent, *Algorithms for Minimization without Derivatives*. Englewood Cliffs, NJ: Prentice-Hall, 1973.
[4] D. Carney et. al., "Monitoring streams: A new class of data management applications," in *VLDB*, 2002.
[5] A. Deshpande and S. Madden, "MauveDB: supporting model-based user views in database systems," in *SIGMOD*, 2006.
[6] "Monthly TAQ, http://www.nysedata.com/nysedata/," New York Stock Exchange, Inc.
[7] U.S. Coast Guard Navigation Center, "Automatic identification system, http://www.navcen.uscg.gov/enav/ais/default.htm."
[8] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos, "Fast subsequence matching in time-series databases." in *SIGMOD*, 1994.
[9] S. Grumbach, P. Rigaux, M. Scholl, and L. Segoufin, "The DEDALE prototype." in *Constraint Databases*, 2000, pp. 365–382.
[10] S. Grumbach, P. Rigaux, and L. Segoufin, "Manipulating interpolated data is easier than you thought." in *VLDB*, 2000.
[11] A. Jain, E. Y. Chang, and Y.-F. Wang, "Adaptive stream resource management using kalman filters," in *SIGMOD*, 2004.
[12] P. C. Kanellakis, G. M. Kuper, and P. Z. Revesz, "Constraint query languages." *J. Comp. and Sys. Sci.*, vol. 51, no. 1, pp. 26–52, 1995.
[13] E. J. Keogh, S. Chu, D. Hart, and M. J. Pazzani, "An online algorithm for segmenting time series," in *ICDM*, 2001.
[14] G. M. Kuper, L. Libkin, and J. Paredaens, Eds., *Constraint Databases*. Springer, 2000.
[15] L.Lin, T.Risch, "Querying continuous time sequences." in *VLDB*, 1998.
[16] N. Mamoulis and D. Papadias, "Slot index spatial join," *IEEE TKDE*, vol. 15, no. 1, 2003.
[17] M. F. Mokbel, T. M. Ghanem, and W. G. Aref, "Spatio-temporal access methods." *IEEE Data Engineering Bulletin*, vol. 26, no. 2, 2003.
[18] R. Motwani et. al., "Query processing, approximation, and resource management in a data stream management system," in *CIDR*, 2003.
[19] C. Olston, J. Jiang, and J. Widom, "Adaptive filters for continuous queries over distributed data streams," in *SIGMOD*, 2003.
[20] L. Neugebauer, "Optimization and evaluation of database queries including embedded interpolation procedures," in *SIGMOD*, 1991.
[21] S. Papadimitriou, J. Sun, and C. Faloutsos, "Streaming pattern discovery in multiple time-series." in *VLDB*, 2005.
[22] H. Shatkay and S. B. Zdonik, "Approximate queries and representations for large data sequences," in *ICDE*, 1996.
[23] M. L. Yiu, N. Mamoulis, and D. Papadias, "Aggregate nearest neighbor queries in road networks." *IEEE TKDE*, vol. 17, no. 6, 2005.