Declarative Temporal Data Models for Sensor-Driven Query Processing

Yanif Ahmad, Uğur Çetintemel
Brown University
{yna, ugur}@cs.brown.edu

ABSTRACT

Many sensor network applications monitor *continuous* phenomena by sampling, and fit time-varying models that capture the phenomena's behaviors. We introduce Pulse, a framework for processing continuous queries over these continuous-time data models. Pulse allows users to declaratively specify both their queries and models, and transforms these queries into simultaneous equation systems, which in many cases are significantly cheaper to process than a stream of discrete tuples. Pulse is able to guarantee user-defined error bounds between query results from continuous-time data models and sampled data, including cases of null results. We present a high-level overview of the design and architecture of Pulse and propose several query optimization techniques that are novel to our context, such as the simplification of our equation systems. We also discuss our plans for extending Pulse to support several novel model types, including differential equations and time series, and outline an abstraction to support query processing on these classes of models.

1. INTRODUCTION

Sensing devices sample physical processes to capture their changing behavior over time. The sensor network and database communities have argued for relational processing of these discrete samples in both sensor databases and stream processing engines. We argue that current processing techniques do not capture that physical processes are continuous, that they exist at every point in time regardless of the sampling mechanism and should be queried as such. Research in the natural sciences relies heavily on data analysis tools in deriving models to compactly represent and accurately reconstruct the behavior of these continuous-time processes. We adopt the position that stream processing engines should support the declarative specification of input attributes as continuous-time models, and should leverage the structure of these models to improve query processing efficiency.

In this paper, we introduce Pulse, a stream processing framework that is capable of applying relational operators, including filters, joins and aggregates, on input streams with attributes represented as polynomials of a time variable. Continuous-time polynomials are a simple class of algebraic models that have been used to approximate many physical processes, such as the temperature of a specific region, the wind speeds observed by a weather station, and the trajectory of cars on a highway segment. Pulse transforms a regular continuous query operating on discrete tuples to simultaneous polynomial equation systems operating on continuous-time models that it can solve efficiently, to reduce computation overhead and improve system throughput. Pulse is able to guarantee error bounds in the approximation provided by the polynomial inputs relative to actual sensor samples. Pulse enables users to specify error bounds at query outputs, and inverts these to corresponding bounds that may be validated at query inputs for efficient bound checking.

We perceive that models of data attributes *constrain* the input data, in a similar fashion to the constraint databases and constraint programming fields. We propose to explore these directions by first extending Pulse's transformation to produce a single global equation system for the whole query rather than operator-centric equation systems. We may then apply static and run-time redundancy elimination techniques to produce a simpler equation system.

We outline initial work on extending Pulse's expressive capabilities to support other types of attribute models such as differential equations and time series. We propose integrating a computer algebra system to manipulate the internal mathematical representation of these datatypes. These manipulations are critical in determining the feasibility of analytical solutions to our equation systems. While analytic methods will be more likely to provide clear gains in processing overheads, we also consider numerical methods for scenarios where analytic methods are not viable. We present a datatype interface, based around the observation that all model types are capable of being *sampled*, and discuss plans to build a common numerical method for performing query processing across different model datatypes.

The remainder of the paper is laid out as follows. Section 2 briefly describes Pulse's basic system model, including the data, query and result models, and a high-level overview of the current architecture. Section 3 discusses our proposals for optimizing Pulse's query processing capabilities. Section 4 describes our goals of extending Pulse's support for different model types. We note related works and techniques in Section 5, before concluding.

2. PULSE OVERVIEW

In this section we present an overview of Pulse, primarily covering the application capabilities it provides in terms of its data stream model, its support for queries and the semantics of query results, and its basic architecture.

This work has been supported in part by the National Science Foundation under grants IIS-0325838 and IIS-0448284.

Proceedings of the 4th International Workshop on Data Management for Sensor Networks (DMSN'07), Vienna, Austria, 2007.

Schema: $A = \{x, v\}, B = \{y, v, a\}$

Query: `SELECT A.x within [-`ϵ`,`ϵ`]`
`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

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

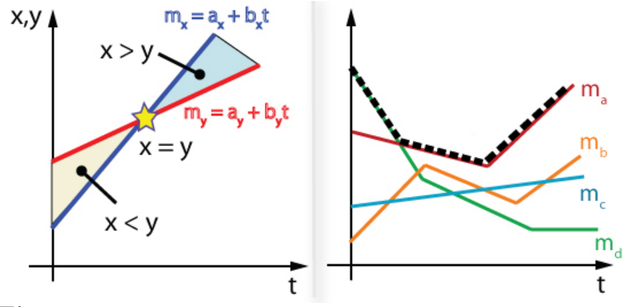

Figure 2: A geometric interpretation of the continuous transform, illustrating predicate relationships between models for selective operators, and piecewise composition of individual models representing the continuous internal state of a `max` aggregate.

2.1 Data Stream Model

We classify data stream attributes as one of modelled attributes, unmodelled attributes, or key attributes. Our framework supports two modelling modes, one where we assume data is modelled externally and specified to Pulse by `MODEL`-clauses, as illustrated in Figure 1. In this mode, we assume model parameters are passed into the system with input tuples and refer to this as *online*, or *predictive* processing. The second mode assumes internal modelling, where we model the data ourselves through a regression operator for *offline*, or *historical* processing. We assume attribute models are polynomial functions of a time variable, for example a model for an attribute a is of the form: $a = \sum_i^d c_i t^i$. Here c_i are model coefficients that are assumed to be present in the input stream. In addition, we assume that users explicitly identify key attributes in the stream declaration and that attributes not defined as keys or models are unmodelled.

Our models are piecewise polynomials, made up of segments, defined as a pair of temporal attributes representing a validity range, along with model coefficients for that range: $([t_l, t_u], \{c_i\})$. We instantiate segments from input tuples, with the valid time range assigned depending on the modelling mode. We consider an update model for segments, allowing consecutive segments to overlap temporally. We assume a default data model for unmodelled attributes, namely that unmodelled attributes are constants throughout a segment's valid time range.

2.2 Query Model

The key contribution of Pulse is that it operates on segments to determine when queries produce outputs by exploiting segments' continuity properties. Pulse provides a declarative query interface by transforming standard stream queries into continuous-time equivalents. We represent queries internally as a system of equations, and define transformations for standard query operators such as filters, joins and aggregates into these equation systems. The solutions to these equation systems yield the time intervals at which predicates are satisfied in selective operators (i.e. filters and joins), or there is a change in the internal state of an aggregate (e.g. min/max aggregates). Correspondingly, the lack of a solution implies a null intermediate result. Our query model also supports key attribute and unmodelled attribute processing, but we omit their details due to space constraints. We now describe the equation systems. Filter and join operators apply predicates defining constraints on the relationships of the input relations' attributes. We assume this is of the form: $x_1 R_1 y_1 \wedge \ldots \wedge x_m R_m y_m$. Given polynomial modelled attributes, we may apply three transformation steps to construct an equation system:

1. Rewrite in difference form: $\forall i. x_i - y_i R_i 0$
2. Substitute attribute models: $\forall i. x_i(t) - y_i(t) R_i 0$
3. Factorize continuous-time functions: $\forall i. (x_i - y_i)(t) R_i 0$

The above predicate defines the following equation system:

$$\begin{bmatrix} x_1^0 - y_1^0 & \ldots & x_1^d - y_1^d \\ \vdots & & \\ x_m^0 - y_m^0 & \ldots & x_m^d - y_m^d \end{bmatrix} \begin{bmatrix} t^0 \\ \vdots \\ t^d \end{bmatrix} \begin{bmatrix} R_1 \\ \vdots \\ R_m \end{bmatrix} \mathbf{0} = \mathbf{D_{xy} t \; R \; 0}$$

Here $\mathbf{t}$ is the continuous-time variable parameterizing attribute models, and $\mathbf{R}$ a vector of relational operators. By solving for this variable, we are able to find intervals for which the predicate is satisfied, providing the input data fits its prescribed model. Figure 2 provides a geometric interpretation of a predicate on a polynomial.

We propose a similar transformation for aggregate operators. The key intuition here is that aggregates maintain internal state, that is only updated on select input tuples for some aggregation functions (e.g. min/max). Thus our equation systems determine when these state updates occur. These types of aggregates compare the input and internal state, for example a min aggregate compares the input with the minimum value seen within a window. This defines an update predicate: $x_1 R_1 s_1 \wedge \ldots \wedge x_m R_m s_m$ where s_i represent internal state attributes. Figure 2 illustrates the internal state attribute as a piecewise polynomial for a max aggregate. This predicate defines an equation system similar to the one above. For non-selective aggregates, such as sum, we propose the computation of window functions that perform temporal aggregation. These window functions are continuous functions, parameterized by a window endpoint and a window length, that compute aggregate values by using equivalent continuous aggregates (e.g. an integration operator for sum aggregates).

2.3 Query Result Model

Pulse produces discrete results from its query processing, preserving client interfaces dependent on query results, by discretizating continuous-time models via sampling at the output streams. We require user-defined sampling rates to drive this discretization, but briefly note that in some cases these may be inferred from the query, for example for sliding window aggregates, the sampling rate is defined by the slide factor. Pulse also provides a mechanism for users to express error bounds in query results, which in turn are used internally to ensure that queries are only processed when current query results are not sufficiently accurate. Pulse supports both absolute errors expressed as ranges, and relative errors expressed relative to results computed from models, and ensures that the differences in continuous and discrete query processing results satisfy these bounds. Given the definition of a bound at query outputs, Pulse inverts the bound to the query's inputs. This enables bounds to be checked entirely at query inputs and eliminates the need to perform discrete processing on input tuples. However inverting bounds limits our technique to invertible expressions, and requires special handling for non-invertible operators such as joins and aggregates. For these non-invertible operators, we explicitly maintain query lineage that associates intermediate results with their input segments, subsequently allowing us to distinguish inputs when inverting bounds. Error bounds are only well-defined in terms of actual query outputs, requiring us to explicitly handle the case of nulls in intermediate results. We define *slack* as a metric that indicates a solver's proximity to producing a solution when an input model causes nulls.

2.4 Pulse Architecture

Pulse transforms each individual operator to create a continuous-time query composed of equation systems that consumes segments and produces segments. Pulse solves its equation systems for every new input model. Here a new input model refers to new values for the attribute models' coefficients (i.e. the set $\{c_i\}$ for any attribute) rather than the symbolic form of the model. We assume the symbolic form of the model remains constant throughout query execution. We have implemented Pulse in Borealis [1], extending the core operator set with several custom operators to implement the systems of equations described above.

Continuous-Time Operators. The two key operator abstractions in Pulse's implementation are the differencer and solver operators. The differencer computes the difference coefficients $\mathbf{D_{xy}}$ defined in Section 2.2, and is implemented as a map or join operator depending on the number of input streams belonging to the transformed operator. In the case of aggregate transformations, the differencer is paired with a stateful operator capable of maintaining an aggregate of multiple segments for the min,max functions, and computing window functions for the sum,avg functions. The solver operator takes a difference coefficient matrix as its input, and computes interval solutions to our polynomial equation system. Our solving method finds polynomial roots, and computes the relevant intervals between root pairs by testing midpoint values according to the equation's predicate. We then filter intervals to common solutions across all equations. In the presence of no solutions, denoting null intermediate results, we pass the difference coefficient matrix to the query inverter component to compute slack. Figure 3 provides a high-level overview of this dataflow.

Error Validation via Query Inversion. Pulse's implementation includes a query inverter for managing the approximation provided by models. The inverter's role is to compute a set of operator input bounds that preserve operator output bounds. We instantiate an inverter for each operator in the original stream query to perform query inversion on a per-operator basis. Inverters explicitly maintain its associated differencer's input segments, and *split* bounds from upstream inverters onto these input segments. We use heuristic-driven split operations to apportion bounds across

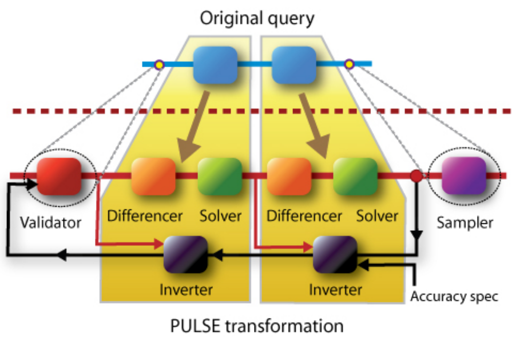

Figure 3: Pulse's continuous-time processor component's dataflow. Models are either determined internally or given as inputs to the system, and are processed as first-class elements.

multiple input segments for inverting the functionality of joins and aggregate operations. We compute slack in the presence of nulls as intermediate results. This requires computing the minimal value of the difference coefficients within the segment as the slack, and initiating a similar inversion process to that of query result inversion. For polynomials, computing this minimal requires taking a derivative and finding a solution where the derivative is zero (and the second derivative is positive). We instantiate a minimizer operator to compute slack for each transformed operator. These components are also illustrated in the dataflow in Figure 3.

Processing Efficiency. We briefly present a experiment from Pulse's current implementation using a moving-object query on a naval vessel dataset containing latitudes and longitudes obtained from the Automatic Identification System (AIS). Our query attempts to detect vessels following one another by checking their proximity over a period of time:

Schema: S = {vessel id, time, lat, lon, lat speed, lon speed}
Query: find neighbouring vessels over a given window

```
select Candidates.id1, Candidates.id2,
  avg(dist) within [-1%,1%]
  from
  (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))
  as Candidates[size 600 advance 10]
group by id1, id2 having avg(dist) < 1000
```

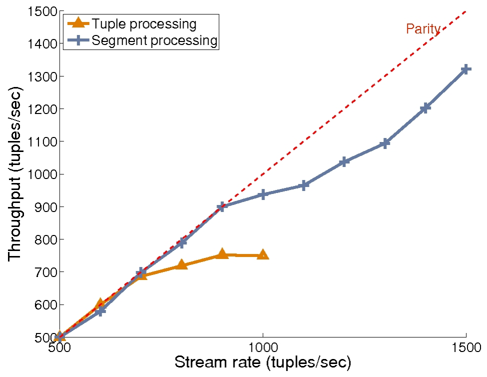

Figure 4: Pulse's processing efficiency on a naval vessel moving-object application.

Figure 4 shows Pulse's end-to-end throughput on the AIS application, for both continuous (segment) processing and discrete (tuple) processing. We see that Pulse scales past a throughput of 700 tuples/sec, where standard stream processing exhausts system capacity, and is able to nearly double throughput at approximately 1400 tuples/sec.

3. QUERY OPTIMIZATION

The focus of these previous sections have been to motivate the benefits of factoring in polynomial attributes into query processing and deriving an initial processing strategy. We now refine this processing strategy by considering optimization opportunities unique to our context.

3.1 Whole-Query Optimization

Whole-Query Linear System Model. We presented Pulse as an operator-by-operator translation from a standard stream operator into a continuous-time operator. This clearly ignores any inter-operator optimization opportunities, such as operator reordering techniques and mechanisms to exploit sharing amongst operators. We define a single system of equations representing the query, from the composition of individual operators' equation systems. This may be accomplished by stacking each equation system and relational predicate. We briefly remark on several properties of this global equation system. First, the dataflow between query operators is implicitly present in the equation system as dependencies between different equations, specifically the coefficients of one equation may be defined as functions of another equation's coefficients, implying our global equation system is not full row rank. Some operators are defined entirely as coefficient operations, for example a sum aggregate.

Coefficient and Constraint Elimination. The global equation system is also an equation system of a single variable, namely the time variable. Thus each equation may be viewed as performing a filtering of the time domain, restricting the set of values the time variable may take on. This motivates our first optimization mechanism, where we attempt to detect subsumption between equations and eliminate these redundancies. We consider two stages where we may detect subsumption relationships. First we apply a static approach where we eliminate equations subsumed symbolically by others in terms of their coefficient expressions. This is effective in the presence of numerous query operators that are dependent on similar input attributes. We perform the static analysis prior to query evaluation, incurring a one-time cost. The second stage of detecting subsumption occurs at run-time, where we inspect the numerical form of equations whenever their coefficient expressions are re-evaluated. Recall that coefficient expressions are evaluated with every model invalidation (due to erroneous modelling). Thus we maintain an index structure for comparing the numerical forms of equations to eliminate redundancies upon updates to arbitrary equations. We envisage an index that supports a lookup method to determine if a new equation is subsumed by any existing equation.

3.2 Scheduling Continuous-Time Queries

In addition to optimizing queries based on their structure, we consider optimizing the evaluation strategies for continuous queries in our stream processing engine.

Equation Scheduling. One challenge we face is that of excessive intermediate segmentation during query processing. This issue is particularly important for stream operators with multiple inputs (joins and unions) or multiple models (aggregates over multiple keys) due to the interaction of different stream segmentations. We propose batching to exploit redundancy due to updates in input segments, and commonality in output segments from unnecessary segmentation (such as with semi-joins, and min/max aggregate operators). We envision a statistical solution to deciding batching characteristics such as how many inputs and outputs to batch. We rely on the average update interval on a stream for determining input batching durations, and the average segmentation of a single input model for output batching durations. This reduction in the intermediate segmentation will provide a tradeoff between our engine's throughput, and the latency of producing query results.

Dynamic Filtering Plans. Following equation elimination, we solve each equation to filter the time variable. We consider the problem of determining an optimal solving order amongst equations. This problem involves identifying equations that indicate an infeasible solution as early in the evaluation order as possible, to ensure minimal wasted work solving equations. We limit the set of potential orders to those that respect dataflows between equation coefficients. We adopt a cost-based approach to determining the optimal order, considering a metric based on each equation's reduction of the time variable's domain rather than the classical notion of selectivity.

4. EXPRESSIVE MODEL TYPES

We view the polynomials described so far as one type of model, and have described how to support polynomials as a first-class datatype in Pulse. We now focus on challenges in providing database support for other model types such as differential equations and time series, and how these may be represented as datatypes. Our goal here is not to present a formal discussion of expressiveness, nor to provide a comprehensive listing of all factors involved in supporting these datatypes, but to convey initial insights into the problem. We believe modelling datatypes should not be black boxes, rather they should expose sufficient semantics for query optimization, similar to enhanced abstract datatypes [11]. We intend to investigate designing an extensible interface and mechanisms for query optimization as future work. One key challenge is to determine how we may leverage mathematical properties of these datatypes in defining query processing mechanisms and performing query optimization. For example, we consider it critical to determine the feasibility of an analytical solution due to the significantly simplified query processing and improved performance it provides, and envision accomplishing this with the aid of a computer algebra system (CAS) integrated with our stream processor.

4.1 Iterative Solving Abstraction

Our data type interface is designed around two common properties of the aforementioned models. First we remark that each type of model supports a *sampling function* that we may use to drive a numerical solver. Next, we observe that numerical methods and solvers are predominantly iterative processes, and that this iteration is often driven by the semantics of the model, rather than data flow. Thus our abstraction decouples control flow from data flow to enable the iterative behavior required to numerically evaluate models. Our framework exposes the following methods for a datatype to implement (in C++ syntax):

```
union InputValue { Tuple; Window; }
union TimeValue { TimePoint; TimeRange; }
union SolutionValue { Point; Segment; }

model_solver:
  void initModel(InputValue, TimeValue);
  boolean hasSolutions();
  pair<TimeValue, SolutionValue> getSolution();
  SolutionValue getSolution(TimeValue);
  boolean checkSolution(Segment, RelOp)
  Segment finalModel(TimeValue);
```

As part of future work, we intend to investigate both conversion abstractions that allow one model type to be cast to another, in addition to a modelling abstraction, where model types may be built or learned incrementally, although it is unclear how much commonality there is across types in these procedures. With this model-specific solving interface, we are able to define a standard numerical query solver based on the evaluation of difference predicates. Our general solver invokes the methods defined by each model type above, from the following pseudocode:

```
query_solver:
void solve_unary_equation(input):
  solver.init(input)
  while ( solver.hasSolutions() )
    solnTime, solnVal = solver.getSolution()
    if ( predicate(solnVal) )
      result = solver.final(solnTime)
      emit(solnTime, result)
    terminate()

void solve_binary_equation(inputA, inputB):
  if solverA.hasDifference(solverB)
    solve_unary_equation(
      solverA.difference(inputA, inputB))
  else
    solverA.init(inputA), solverB.init(inputB)
    while ( solverA.hasSolutions() )
      solnATime, solnAVal = solverA.getSolution()
      solnBTime, solnBVal = solverB.getSolution(solnATime)
      if ( predicate(solnAVal, solnBVal) )
        result = output_attribute(
          solverA.final(solnBTime),
          solverB.final(solnBTime))
        emit(solnBTime, result)
      terminate()
```

The above example shows symbolic manipulations with the `hasDifference` and `difference` functions. We show a difference equation with two variables being solved as a difference equation of one variable by computing a single model (segment) representing that difference. This solver also highlights other interesting challenges, including synchronization (note `solverB` samples at the time value yielded by `solverA`) and solver termination policies. Solver termination concerns how an optimizer might take advantage of mathematical properties to limit the solutions produced to exclude those that cannot be query results. Processing queries involves evaluating the above methods for each equation in the whole-query equation system representation, and filtering to common solution time ranges where each equation is satisfied. Note our interface is closed, that is our solver produces the same model type as its input.
We now present example interface implementations.

Temporal Polynomials. Figure 5 provides a high-level functional description of an API implementation for Pulse's current polynomial equation system solver, supporting the

```
initModel : set input segment as state
            determine multiplicity of roots
hasSolutions : return if any roots remain
getSolution : return time range and segment
              between last root and current root
checkSolution : check if entire segment
                satisfies relational operator
finalModel : return input segment
```

Figure 5: Example polynomial type sampling function.

modelling of attributes as continuous functions of a time variable. A key property of this type of model is that we are able to simplify the internal representation of the model for the basic arithmetic operators $+, -, *$ (and in some cases $/$).

4.2 Differential Equations

We consider stream attributes represented by n-th order ordinary, linear differential equations of the form: $D^n x_i = \sum_j^{n-1} c_j(x_i) D^j x_i$, where D is the differential operator. We provide a simple example of a query specifying an input attribute as a differential equation, capturing a simple carbon dating application. Here the number of carbon-14 molecules in a material is modelled by a differential equation governing the rate of decay of these molecules. We omit the error bound specifiers for presentation clarity in the remaining examples in this section.

Schema: carbons = {material, amount}
Query: find when #carbon-14 isotopes drops below a threshold, for each material.
Model: $dn/dt = -\lambda n$, where n is the #carbon-14 isotopes, and λ the decay constant.

```
select material, t, amount
from carbons model
  d(amount)/d(t) = - λ * amount
where amount < c
```

Figure 6 briefly describes the implementation of a solver performing a numerical approximation of an ordinary differential equation. In terms of interface design, the critical difference between the solver requirements for this type of model and polynomial models is that we are only able to sequentially compute solutions, where we are provided with an initial value satisfying the equation, and numerically approximate subsequent values from previous approximations of derivatives.

```
initModel : i) set input segment as coefficients
            and initial value
            ii) set time range start as eval point
hasSolutions : i) check if eval point is at
               time range end
               ii) check for more solution ranges given
               solving termination policy
getSolution : i) evaluate an Euler integration step
              updating derivative coefficients
              ii) advance eval point
checkSolution : floating point comparison of solutions
finalModel : return input segment
```

Figure 6: Example differential equation solver.

We have only discussed a small subset of the many classes of differential equations that are used in real-world applications. Other common types include stiff differential equations, and more complex partial differential equations. Due to their prevalence in the physical sciences, there are a large number of high-quality implementations of differential equation solvers, including commercial tools such as Matlab, and opensource tools such as SUNDIALS [7]. We believe it is critical from a software engineering perspective to reuse these tools, subject to a match of the tools' interfaces to ours (for example we could use an off-the-shelf solver to find all solutions within the `initModel` function, but this would unnecessarily generate solutions).

4.3 Time Series Models

We consider the standard ARMA, ARIMA and SARIMA time-series models, and assume we are given the coefficients of the various components of these models. For example, an attribute x represented by an ARMA(w,q) model is given by: $x_t = \sum_i^w a_i x_{t-i} + \sum_i^q b_i \epsilon_{t-i}$, where $\{a_i\}, \{b_i\}$ are coefficients present in the input streams allowing us to internally construct a time series datatype. Many sensor devices generate time series data, and we present an example of a continuous query monitoring seismometers to detect earthquakes near densely populated regions, as found in disaster management applications such as GDACS [4].

Schemas: Population = {lat, lon, population, needs index}
Seismometer, S = {time, lat, lon, amplitude, a_1, a_2, b_1, b_2}
Query: find significant earthquakes near population centers
Models: $S.ampl = \sum_i S.a_i \times S.ampl_{t-i} + \sum_i S.b_i \times \epsilon_{t-i}$

```
select lat, lon, mag from
(select alert_score(pop, mag, gn), mag, P.lat, P.lon from
  (select lat, lon, max(amplitude) as mag from
    stream S[size 10 advance 2]
    model S.amplitude = timeseries(ar(a1, a2), ma(b1, b2))
    group by lat, lon)
  as EQ join Population P
  on (dist(EQ.lat, EQ.long, P.lat, P.lon) < 100km))
as PotentialAlerts
where alert_score > 2 and mag > 6
```

Figure 7: Earthquake detection on seismic data represented by time series models.

In a similar manner to differential equations, evaluating time series requires that we use a window of existing attribute values to compute the next value. This limits time series evaluation to generating a single point per model. We use a time series within its associated valid time range by using generated points themselves as part of the window. Furthermore, the nature of an autoregressive process is to use every input tuple to compute coefficients, raising an interesting question for future work of how we may combine a modelling step and solving step with each iteration.

5. RELATED WORK

Recently, there has been significant interest in supporting modelling techniques in database systems for purposes ranging from data cleaning [9], regression and interpolation [2], to predicting inputs for query processing [8]. Scientific databases have consistently looked at various types of data including arrays [10], and time series and spectra [13]. Girod et al. [3] recently presented a stream processing architecture focusing on systems issues and programmatic support for signals as a first-class datatype in the Wavescope system. Time series databases provide basic functionality to construct and add samples to time series, in addition to providing query functionality [14] that differs quite significantly from the relational query processing we target. Moving object databases have developed highly customized algorithms and data structures to support query processing over trajectories [6]. Finally, constraint databases such as DEDALE [5] have considered representing relations as constraints, and processing queries via a constraint engine. Srivastava [12] describes techniques to exploit subsumption and indexing in constraint databases. While these works have inspired us, we are primarily interested in understanding the basic challenges involved in relational query processing of time-varying data types, and the uses of these data types in historical and predictive applications. Our contribution is that we have developed a general stream processing framework that is capable of supporting core relational operators in a common representation as simultaneous equation systems. Furthermore, related works have not studied the problems such systems face under the online detection of errors, nor do they extend to other model types.

6. CONCLUSION

We have outlined Pulse, a stream processing framework designed to handle declarative specifications of both queries, and the time-varying behavior in query inputs. The novel features of Pulse includes its behind-the-scenes transformation of declarative queries into simultaneous equation systems and query processing error validation at the input streams. We have implemented these features of Pulse in Borealis [1]. Now, we propose several query optimization techniques including representing a whole query as a single equation system, rather than a per-operator structure. We also consider the question of how to support model types such as differential equations and time series, and briefly outlined a datatype abstraction that we hope to use in a common numerical solving method across model types.

7. REFERENCES

[1] D. Abadi et. al. The design of the Borealis stream processing engine. In *CIDR*, 2005.

[2] A. Deshpande and S. Madden. MauveDB: supporting model-based user views in database systems. In *SIGMOD*, 2006.

[3] L. Girod et. al. The case for a signal-oriented data stream management system. In *CIDR*, 2007.

[4] Global Disaster Alert and Coordination System (GDACS). http://www.gdacs.org/.

[5] S. Grumbach, P. Rigaux, M. Scholl, and L. Segoufin. The DEDALE prototype. In *Constraint Databases*, 2000.

[6] M. Hadjieleftheriou et. al. Complex spatio-temporal pattern queries. In *VLDB*, 2005.

[7] A. C. Hindmarsh et. al. Sundials: Suite of nonlinear and differential/algebraic equation solvers. *ACM Trans. Math. Softw.*, 31(3), 2005.

[8] A. Jain, E. Chang and Y. Wang. Adaptive stream resource management using Kalman Filters. In *SIGMOD*, 2004.

[9] S. R. Jeffery, M. N. Garofalakis, and M. J. Franklin. Adaptive cleaning for rfid data streams. In *VLDB*, 2006.

[10] A. P. Marathe and K. Salem. A language for manipulating arrays. In *VLDB*, 1997.

[11] P. Seshadri. Enhanced abstract data types in object-relational databases. *VLDB J.*, 7(3), 1998.

[12] D. Srivastava. Subsumption and indexing in constraint query languages with linear arithmetic constraints. *Ann. Math. Artif. Intell.*, 8(3-4), 1993.

[13] R. H. Wolniewicz and G. Graefe. Algebraic optimization of computations over scientific databases. In *VLDB*, 1993.

[14] Y. Zhu and D. Shasha. Warping indexes with envelope transforms for query by humming. In *SIGMOD*, 2003.