Querying Continuous Functions in a Database System

Arvind Thiagarajan
MIT CSAIL
Cambridge, MA, USA
arvindt@csail.mit.edu

Samuel Madden
MIT CSAIL
Cambridge, MA, USA
madden@csail.mit.edu

ABSTRACT

Many scientific, financial, data mining and sensor network applications need to work with *continuous*, rather than discrete data *e.g.*, temperature as a function of location, or stock prices or vehicle trajectories as a function of time. Querying raw or discrete data is unsatisfactory for these applications – *e.g.*, in a sensor network, it is necessary to interpolate sensor readings to predict values at locations where sensors are not deployed. In other situations, raw data can be inaccurate owing to measurement errors, and it is useful to fit continuous functions to raw data and query the functions, rather than raw data itself – *e.g.*, fitting a smooth curve to noisy sensor readings, or a smooth trajectory to GPS data containing gaps or outliers. Existing databases do not support storing or querying continuous functions, short of brute-force discretization of functions into a collection of tuples. We present FunctionDB, a novel database system that treats mathematical functions as first-class citizens that can be queried like traditional relations. The key contribution of FunctionDB is an efficient and accurate *algebraic* query processor – for the broad class of multi-variable polynomial functions, FunctionDB executes queries directly on the algebraic representation of functions without materializing them into discrete points, using symbolic operations: zero finding, variable substitution, and integration. Even when closed form solutions are intractable, FunctionDB leverages symbolic approximation operations to improve performance. We evaluate FunctionDB on real data sets from a temperature sensor network, and on traffic traces from Boston roads. We show that operating in the functional domain has substantial advantages in terms of accuracy (15-30%) and up to order of magnitude (10x-100x) performance wins over existing approaches that represent models as discrete collections of points.

Categories and Subject Descriptors

H.2.4 [**Information Systems**]: Database Management — Systems — Query processing; H.2.4 [**Information Systems**]: Database Management — Languages — Query languages; I.1.4 [**Symbolic And Algebraic Manipulation**]: Applications

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD'08, June 9–12, 2008, Vancouver, BC, Canada.
Copyright 2008 ACM 978-1-60558-102-6/08/06 ...$5.00

General Terms

Algorithms, Languages, Experimentation, Performance

1. INTRODUCTION

Relational databases have traditionally taken the view that the data they store is a set of discrete observations. This is reasonable when storing individual facts, such as the salary of an employee or the description of a product. However, when representing time- or space-varying data, such as a series of temperature observations, the trajectory of a moving object, or a history of salaries over time, a set of discrete points is often neither the most intuitive nor compact representation. Indeed, for researchers in many fields – from social sciences to biology to computer science – a common first step in understanding a set of data points is to model those points as a collection of mathematical functions. In some situations, continuous functions emerge naturally from the data – for example, a vehicle trajectory can be represented as a series of road segments derived from geographic data. In others, functions are generated from discrete data using some form of regression (curve fitting). In many of these applications, posing queries in the curve or function domain yields more natural and/or accurate answers compared to querying raw data, for one or both of the following reasons:

Continuous Data. For many applications, a set of discrete points is an *inherently* incomplete representation of the data. For example, if a number of sensors are used to monitor temperature in a region, it is necessary to interpolate sensor readings to predict temperature at locations where sensors are not physically deployed.

Noisy Data. Raw data can also contain measurement errors, in which case simple interpolation does not work. For example, in a sensor network, sensors can occasionally fail, or malfunction and report garbage values due to low batteries/other anomalies. In such situations, it is preferable to query values predicted by a regression function fit to the data, rather than the raw data itself.

The problem of querying continuous functions is not adequately addressed by existing tools. Packages like MATLAB support fitting data with regression functions, but do not support declarative queries. Databases excel at querying discrete data, but do not support functions as first-class objects. They force users to store continuous data as a set of discrete points, which has two drawbacks – it fails to provide accurate answers for some queries, as discussed, and for many other queries, it is inefficient because a large number of discrete points are required for accurate results.

This paper describes FunctionDB, a database system that allows users to query the data represented by a set of continuous functions. By pushing support for functions into the database, rather than requiring an external tool like MATLAB, users can manage these models just like any other data, providing the benefits of declarative queries and integration with existing database data. FunctionDB supports special tables that can store mathematical functions, in addition to standard tables with discrete data. The system provides users with tools to input functions, either by fitting raw data using regression, or directly if data is already continuous (*e.g.*, road segments). For example, a user might use FunctionDB to represent the points $(t = 1, x = 5), (t = 2, x = 7), (t = 3, x = 9)$ as the function $x(t) = 2t + 3$, defined over the domain $t \in [1, 3]$.

In addition to creating and storing functions, FunctionDB allows users to pose familiar relational queries over the data represented by these functions. The key novel feature of FunctionDB is an *algebraic* query processor that executes relational queries using symbolic algebra whenever possible, rather than converting functions to discrete points for query evaluation. Relational operations become algebraic manipulations in the functional domain. For example, a selection query that finds the time the temperature of a sensor whose value is described by the equation $x(t) = 2t + 3$ crosses $x = 5$ requires solving the equation $2t + 3 = 5$ to find $t = 1$. We describe similar symbolic analogs for aggregation and join in this paper.

Algebraic query processing is challenging because queries over functions of multiple variables can generate systems of inequalities for which closed form solutions are intractable. Consider `temp` $= f(x, y)$, a model of temperature as a function of x and y location, and suppose a user writes a query like `SELECT AVG(temp) WHERE` $x^2 + y^2 < 20$, which requests the average temperature in a circular region. This requires computing the integral of $f(x, y)$ over the region. For arbitrary functions, or regions defined by arbitrary `WHERE` predicates, closed form solutions either do not exist, or are expensive to compute. Hence, FunctionDB chooses a middle ground between brute-force discretization and a fully symbolic system. We show how to efficiently approximate a broad class of regions – those defined by arbitrary systems of *polynomial* constraints (such as the region in the `WHERE` clause above) by a collection of hypercubes. We evaluate queries over the original region by combining results of closed-form evaluation over each of the hypercubes.

FunctionDB is related to constraint databases and constraint programming systems [8,10,15], but in contrast to these systems, which typically strive to find closed form solutions, and are hence slow or intractable for nonlinear constraints, FunctionDB queries return *discrete* relational-style results intuitive to an end user. Hence, we can leverage approximation to achieve good performance for a wide class of polynomials.

Another closely related system is MauveDB [4], which stores models as discrete points that are processed by traditional relational operators *e.g.*, the function $y(x) = 2x + 1$ might be stored and queried as the set $(0, 1), (1, 3), (2, 5) \ldots$. Although FunctionDB uses gridding for query answers, it grids data as late as possible in query plans. Query processing happens primarily over ungridded data, yielding substantial efficiency and accuracy gains over MauveDB.

In summary, this paper makes two technical contributions. The first is a sound data model supporting queries over collections of piecewise polynomial functions. The second is an algebraic query processor which operates as far as possible on the symbolic representation of functions without materializing them into discrete points. We describe relational operators as a combination of algebraic primitives: function evaluation, equation solving, function inference, variable substitution and symbolic integration. We also show how to efficiently evaluate queries over regions enclosed by polynomial constraints by approximating them with hypercubes, while preserving the semantics of query results. We evaluate FunctionDB on two real data sets: data from 54 temperature sensors in an indoor deployment, and traffic traces from cars. FunctionDB achieves order of magnitude (10x-100x) better performance for aggregate queries and 2-10x savings for selective queries, compared to MauveDB-like approaches that represent functions as discrete points. FunctionDB is also more accurate than gridding, which results in up to 15-30% discretization error. We evaluate our approximation strategy for multiple-variable constraints on the temperature data set, and show that for selective queries, it performs up to 10x faster than gridding.

2. OVERVIEW AND QUERY LANGUAGE

The key abstraction provided by FunctionDB is the *function view*, a logical interface to continuous functions which is virtually identical to querying raw data (similar to a model-based view [4]). For example, a function view of temperature as a function of (x,y) location would appear logically as a table with 3 attributes: x, y and temperature, even though the underlying physical representation is a set of mathematical functions expressing temperature in terms of x and y in different regions. Users can pose normal relational queries (filters/joins/aggregates) over these views, or join them with regular relational tables. FunctionDB translates these queries to algebraic operations over the underlying functions. This section discusses the DDL (data definition language) used to create function views, and the query language used to pose queries over these views. We use examples from two real-world applications: an indoor sensor network and an application that analyzes car trajectory data.

Creating Function Views (DDL). A function view can be created either by fitting a regression curve to raw data, or imported directly from functional data, in both cases using a `CREATE VIEW` statement. We first discuss the regression approach in the context of our sensor network application, in which a network of temperature sensors are placed on the floor of a building (we use real data from a lab deployment). Each sensor produces a time series of temperature observations. Fitting a regression curve is useful here for several reasons: first, temperature is only available at select sensor locations, but users would like to know the temperature at all locations, requiring interpolation. Second, radios on sensors lose packets; third, nodes can fail (*e.g.*, due to low batteries), producing no readings or noisy data, which regression helps smooth over.

Regression takes as input a data set with two or more correlated variables (*e.g.*, time and temperature, or location and temperature), and produces as output a formula for one of the variables, the *dependent* variable, as a continuous function of the other variable(s), the *independent* variable(s). The aim of regression is to approximate ground truth with as little error as possible. The simplest form of regression is linear regression, which takes a set of basis functions of the independent variables (e.g., x^2, xy, y^2) and computes coefficients for each of the basis functions (e.g., a,b,c) such that the sum of the products of the basis functions and their coefficients

(e.g., $ax^2 + bxy + cy^2$) produces a least-squares fit for an input vector of raw data, X. Performing linear regression is equivalent to performing Gaussian elimination on a matrix of size $|F| \times |F|$, where $|F|$ is the number of basis functions. A standard way to use regression in modeling data is to first *segment* data into pieces within which it exhibits a well-defined trend, and then choose basis functions most appropriate to fit the data in each piece.

We now illustrate the use of `CREATE VIEW`. We assume raw temperatures are stored in a relational table, `tempdata` whose schema is `(ID,x,y,time,temp)` – `ID` is the sensor ID, `(x,y)` are sensor coordinates, `time` is the measurement time, and `temp` is the observed temperature. The following query fits a piecewise linear model of temperature as a function of time to readings in `tempdata`:

```
CREATE VIEW timemodel
AS FIT temp OVER time^1
USING PARTITION FindPeaks, 0.1
TRAINING_DATA SELECT * FROM tempdata
GROUP ON ID, x, y
```

This instructs FunctionDB to fit a function view `timemodel` using `temp` as the dependent variable and `time` as the independent variable. This view can be queried like a relational table with schema `(ID, x, y, temp, time)`. The basis `time^1` in the `OVER` clause determines that the regression fit is a line (degree 1 polynomial). `USING PARTITION` tells FunctionDB how to partition the data into pieces within which to fit different regression functions. Here we use `FindPeaks`, a simple in-built segmentation algorithm that finds peaks and valleys in temperature data and segments the data at these extrema (the knob `0.1` determines how aggressively the algorithm splits data into pieces). The `TRAINING_DATA` clause specifies the data used to train the regression model, in this case all the readings from `tempdata`. The `GROUP ON` clause specifies that different models should be fit to data with different sensor IDs or different locations. The attributes in `GROUP ON` (here `ID`, `x` and `y`) always appear as additional attributes in the view (here `timemodel`).

We used `FindPeaks` purely for illustration: any suitable model fitting strategy can be plugged into our framework. Segmentation and model selection have been researched extensively in the machine learning and data mining literature ([11] has a survey). Also, as pointed out in [4], maintaining models online as new raw data arrives can be crucial for performance. We do not focus on either of these aspects of updates in this paper. Our prototype supports fitting models in a single pass over raw data using linear regression, and simple segmentation strategies like `FindPeaks`. Our focus is on the query language, and on efficient query processing over models post-fit. For a detailed discussion of segmentation/update algorithms, we refer the reader to a related thesis [18].

The following `CREATE VIEW` statement models temperature as a function of 3 variables: `x`, `y` and `time`, and in addition uses quadratic (degree 2) bases for `x` and `y` (terms without a ^ imply degree 1):

```
CREATE VIEW 3dmodel
AS FIT temp OVER x^2, xy, y^2, x, y, time
USING PARTITION RegressionTree
TRAINING_DATA SELECT x, y, time, temp FROM tempdata
```

Here, we use the popular regression tree algorithm [12] as a segmentation strategy in multiple dimensions. A regression tree hierarchically partitions data into a tree where each internal node is a split of the data into segments along some independent variable. For example, a node containing the predicate $x < 4$ would have two branches as children, with the left branch representing the region of space where $x < 4$, and the right branch the region where $x \geq 4$. A path from the root to each leaf identifies a sequence of such splits, and hence a subregion of space. Each leaf specifies a single regression function, which is fit to data within its subregion.

Querying Function Views (QL). FunctionDB's query language is similar to traditional SQL. However, because the underlying physical representation uses functions rather than raw data, query results are more accurate/intuitive. For example, consider the following query, which looks up temperature at a specific location and time using `3dmodel` (defined above):

```
SELECT temp FROM 3dmodel
WHERE x = 20 AND y = 20 AND time = 10      (Query 1)
```

This query returns a temperature even if a sensor is not physically deployed at (20, 20), by evaluating `3dmodel` at (20, 20).

To support querying continuous functions, FunctionDB augments SQL with two key extensions, described below:

GRID clause. FunctionDB represents functions symbolically and executes many queries without materializing functions into discrete points. The results of symbolic operations are *continuous* intervals/regions, not discrete points. Since it is useful to return a finite result, many non-aggregate queries need to include a `GRID` clause specifying the granularity of query results. `GRID` discretizes the independent variables at fixed intervals to generate discrete tuples similar to a traditional DBMS. Below, we give two example queries on `3dmodel` which contain a `GRID` clause:

Locations where temperature is below threshold (*Filter*)

```
SELECT x, y FROM 3dmodel WHERE temp < 18
GRID x 0.5, y 0.5                          (Query 2)
```

Locations reaching thresholds (T_1, T_2) within time `t`, and the times at which these are reached (*Join*)

```
SELECT m1.x, m1.y, m1.time, m2.time
FROM 3dmodel AS m1, 3dmodel AS m2
WHERE m1.x = m2.x AND m1.y = m2.y AND
m1.temp = T1 AND m2.temp = T2 AND
ABS(m1.time- m2.time) < t
GRID x 0.5, y 0.5                          (Query 3)
```

The `GRID x 0.5, y 0.5` in Query 2 specifies that regions where temperature is $< 18^o$ should be output as a grid of (x, y) coordinates at integer multiples of 0.5 along both x and y. However, not all queries require grid sizes to be specified for all the selected columns. Query 1 does not require a `GRID` clause because the query binds all the independent variable values. Query 3 requires `GRID` for only two of the selected columns because the equations in its `WHERE` clause determine finite sets for the other variables without gridding. Section 3 explains how the system performs this inference.

Continuous Aggregates. FunctionDB provides aggregate operators which generalize discrete aggregate operators to operate on continuous functions. Also, `GROUP BY` queries include a `GROUPSIZE` clause when grouping by continuous variables to indicate a bin size for grouping. For example, the following query computes a histogram showing the durations for which a given sensor experiences different temperature ranges:

```
SELECT temp, VOLUME(time) FROM timemodel
WHERE ID = given_id
GROUP BY temp GROUPSIZE 1                    (Query 4)
```

The `GROUPSIZE 1` clause groups temperature into bins of size 1^oC and applies the `VOLUME` aggregate to each bin. `VOLUME` is an aggregate operator which generalizes SQL `COUNT`: it computes the range of a continuous variable rather than the count of a discrete variable. We detail other similar operators in Section 3.

We now show how function views are useful in our second application, which analyzes car trajectory data from Cartel [9], a mobile sensor platform that has been deployed on a number of automobiles. Each car is equipped with an embedded computer connected to several sensors and a GPS device for determining vehicle location in real time. We consider queries on car trajectories specified by a sequence of GPS readings. GPS data requires interpolation because it sometimes has large gaps when the GPS signal is weak *e.g.*, when the car passes under a bridge or through a tunnel. GPS values are also sometimes incorrect or include outliers.

We model GPS data using piecewise linear (degree 1) functions representing road segments. We use this model primarily for illustration (more sophisticated models like splines might be preferable in reality), because it fits the GPS data quite well, and smooths over gaps and errors. The schema of the function view of a car trajectory is `(tid, lon, lat)` where `tid` is a trajectory identifier, and `lat` (latitude) is modeled as a piecewise linear function of the independent variable, `lon` (longitude) on each road segment. As in the temperature example, querying this model is preferable to querying raw data. For example, consider the following query, which counts the number of trajectories passing through a given bounding box:

```
SELECT COUNT DISTINCT(tid) FROM trajectories
WHERE lat > 42.4 AND lat < 42.5 AND
AND lon < -71 AND lon > -71.1                (Query 5)
```

The advantage of using a function view here is that even if the bounding box in this query happens to fall entirely in a gap in the GPS data, the trajectory counts as passing through the box as long as the underlying line segment(s) intersect it.

We next show how to use FunctionDB to implement an interesting application of the Cartel data – finding routes between two locations taking recent road and traffic conditions into account. To do this, we need to cluster actual routes into similar groups, and compute statistics about commute time for each cluster. We express this query as a self-join that finds pairs of trajectories that start and end near each other, computing a similarity metric for each pair. Our metric pairs up points from trajectories corresponding to the same distance *fraction* along their respective routes (*e.g.*, midpoints are paired together, and so are first quartiles). The metric computes the average distance between the points over all pairs. To do this, we first build a view `fracview` from the trajectory data with a precomputed distance fraction, `frac` $\in [0, 1]$. Here, we show a join query over `fracview` that computes all-pairs trajectory similarities:

```
SELECT table1.tid, table2.tid,
AVG(sqrt((table2.lon - table1.lon)^2 +
        (table2.lat - table1.lat)^2))
FROM fracview AS table1, fracview AS table2,
WHERE table1.frac = table2.frac AND
      table1.tid < table2.tid
GROUP BY table1.tid, table2.tid              (Query 6)
```

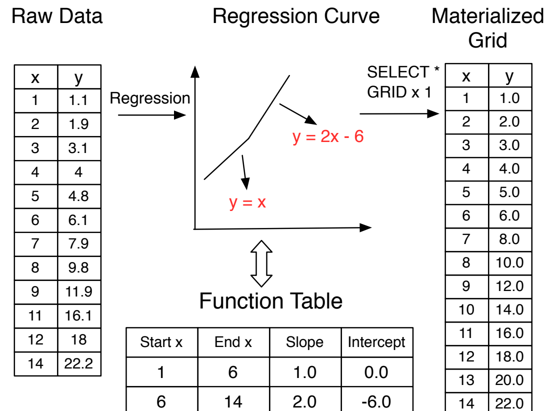

Figure 1: Raw data, function table, and result of `SELECT *`.

Lastly, this application illustrates that functions need not always come from regression. For example, a better alternative to model Cartel data might be to map GPS data to road segments from a geographic database. To import data already in functional form, like road segments, we can extend `CREATE VIEW` to import function views from a relational table containing coefficients.

3. DATA MODEL AND QUERY PLANS

We now describe how FunctionDB represents functions, and discuss a key aspect of our query semantics, different from previous systems that aim to find closed form solutions: all query results are "gridded" *discrete* points sampled at regular intervals. We show how this enables a sound data model into which symbolic operations integrate seamlessly.

3.1 Function Tables

The *logical* data model in FunctionDB from a user perspective is a collection of points with schema identical to the raw data. While functions are used to fit data and answer queries accurately, this happens under the hood. This has the advantage that SQL queries written to work on raw data can run with little or no modification over models. Of course, functions also play a central role in FunctionDB's *physical* data model. The key feature of our physical data model is the *function table*, a table used to store a collection of functions and *constraints* (predicates over functions). While functions can be generated by any mechanism as discussed previously, we use regression as the example from this point.

When generated by a piecewise regression model, rows of a function table represent pieces of the model. Each piece is a continuous function $Y = F(X)$, expressing a dependent variable Y in terms of one or more independent variable(s) X. Each piece is defined over a region which bounds the range of values for the independent variables X – this is an interval in 1D, and a hypercube in higher dimensions. Each piece contains parameters specifying the structure of the function (*e.g.*, polynomial coefficients) in that region. Figure 1 shows a function table defined over data with two attributes x and y. The data has been fit with two functions that express y (dependent variable) as a function of x (independent variable): $y = x$ when $1 \leq x < 6$, and $y = 2x - 6$ when $6 \leq x < 14$. "Start x" and "End x" form the domain; "Slope" and "Intercept" are parameters.

Because functions do not suffice to represent the intermediate re-

sults of queries with WHERE predicates, our physical data model also supports *constraints*. For example, if we want to evaluate the predicate $z < 25$ over the function $z = x^2 + y^2$, we need to represent a 2-dimensional circular region: $x^2 + y^2 < 25$. We permit intermediate tuples in FunctionDB query plans to contain multiple constraints and functions. For example, the tuple ($0 \leq x \leq 5$, $3 \leq y \leq 8$, $z = x^2 + y^2$, $z \leq 25$) represents the intersection of the circular region $x^2 + y^2 \leq 25$ with the rectangle specified by the first two constraints, with z being the dependent variable and x, y the independent variables. In this respect, FunctionDB uses the same physical data model as constraint databases [5, 15], which aim to find closed form solutions to constraint systems. However, as the next section explains, FunctionDB's logical data model, and hence query semantics, are very different from these systems.

To represent non-single-valued "functions" (*e.g.*, car trajectories), we permit overlapping hypercubes in function tables. Such a table represents the union of points over all pieces *e.g.*, a table with $y = x$, $y = x + 1$ both defined over $[0, 1]$ represents the points on two line segments, with two values of y for each $x \in [0, 1]$.

3.2 Query Semantics and Gridding

Although each piece in a function table has a compact representation, logically, it represents an *infinite* set consisting of every single point on a continuous curve/surface representing the function. This is very different from a traditional relation, which is a finite set of tuples, and makes query semantics challenging to formulate. The logical result of the simplest of queries: `SELECT *` over a function table is an infinite collection of tuples. Since it is impossible to output an infinite set, `SELECT *` either needs to output a closed form representation of the set (here, the functions themselves!), or results sampled from the set in a regular way.

A purely closed form approach, as adopted by constraint programming systems [8, 15] has several drawbacks. First, closed form solutions are not always feasible for arbitrary queries. Second, even if a "closed form" solution like $f(x) = 0$ exists, if $f(x)$ is arbitrarily complex, this is not useful if results need to be used further – *e.g.*, displayed to the end user, or joined with tables containing discrete data. Hence, we opt for a middle ground. We keep *intermediate* results in closed form as far as possible, and exploit algebraic operations over these results to improve query efficiency. However, the *final* result of a query is always a well-defined finite set, not a closed form solution. This set is computed by a procedure we term *gridding*, which denotes sampling at regular intervals from the space of all points satisfying the query. Sampling is performed according to a GRID clause specified by the end user. For example, the query `SELECT * GRID x 1` in Figure 1 generates values of `x` from the domain which are integer multiples of 1, evaluates $y = f(x)$ at each grid point, and outputs the resulting (x, y) pairs.

"Grid semantics" has important advantages. First, there is a simple, correct execution strategy (the "gridding strategy") for all queries – sample values for each of the independent variables at intervals of the grid spacing, evaluate each of the functions at each grid point, and evaluate the rest of the query (*e.g.*, filters) using traditional DBMS operators. This works for arbitrarily complex functions, as long as we know how to evaluate a function at any given point. Second, gridding is intuitive from a user perspective – querying a function table generated via regression gives similar results to querying raw data, though more accurate because the model helps correct errors/gaps in raw data. Third, even if a closed form solution does not exist, we can leverage approximation to improve performance, and moreover, tune our approximation to satisfy grid semantics at the required grid size (Section 4.3). This performs considerably better than gridding, and allows the user to control the granularity of results by varying the grid size.

Our grid semantics are somewhat similar to the grid representation in MauveDB [4]. However, MauveDB does no algebraic evaluation, and applies gridding immediately as functions are read into memory, or materializes the results of gridding on disk – in which case the user cannot control the grid size for queries.

3.3 Algebraic Query Plans

The basic gridding strategy is simple and works correctly for the widest class of functions/constraints, but slow because each grid point in the region of definition needs to be processed, and a large number of grid points are required for accurate results for many queries. We now describe significantly faster symbolic query plans for the special, but widely used class of *polynomial* functions. Our plans are based on the standard iterator model of query execution, where operators are connected in a tree, and tuples flow from leaves to the root. We distinguish between *tuples* and *grid points*: tuples are records that can contain functions or constraints. A tuple usually represents an infinite number of grid points.

Because the end goal of all queries (except aggregates) is to produce a gridded result, we require that all algebraic operators satisfy an important invariant: the output tuples they produce must contain a *bounding hypercube* defined over all the independent variables in the tuple. A variable is independent if it is not the RHS of any function. For example, in a function table with 2 variables x and y, a constraint like ($0 \leq x \leq 5$, $3 \leq y \leq 8$) is a bounding hypercube. The bounding hypercube invariant ensures we can grid the output of *any* operator, and build modular plans that use both symbolic operations and gridding.

GRID operator. Gridding is simple and only requires the ability to evaluate functions and constraints. It works by applying three operations to each tuple, in order:

1. Enumerate grid points from the bounding hypercube, located at integer multiples of the grid size along each independent variable.
2. Evaluate the dependent variables at each of the grid points.
3. Discard grid points that do not satisfy constraints in the tuple.

For example, applying this procedure to the tuple: ($0 \leq x \leq 5$, $3 \leq y \leq 8, z = x^2 + y^2, z \leq 25$) using a grid size of 2 for both x and y would yield ($x = 0, y = 4, z = 16$), and ($x = 2, y = 4, z = 20$).

Our query plans use 4 key algebraic operations: variable substitution, equation solving/approximation, function inference, and symbolic integrals. While computer algebra systems like Mathematica employ many other such techniques, we chose these specific techniques because they are simple to implement, fit well with our semantics and give us significant performance benefits. We now illustrate these techniques using a simple selection query:

```
SELECT x,y FROM tempmodel
WHERE temp < T0 GRID x gs, y gs
```

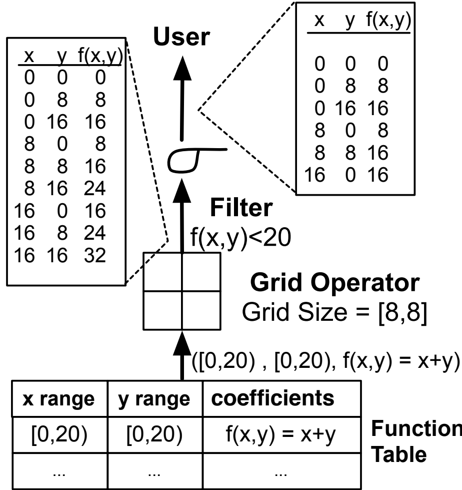

Figure 2: Grid plan for `SELECT x,y WHERE` $f(x, y) < 20$**.**

Here we assume temperature is modeled as a piecewise polynomial function of location, and each entry in the function table `tempmodel` contains two fields: a bounding rectangle over `x` and `y`, and coefficients for the function `temp` = $f(x, y)$ over that rectangle. The user wants to find locations where `temp` falls below a threshold T_0, spaced at intervals of the grid size gs. The gridding strategy to execute this query is shown in Figure 2, for $T_0 = 20$ and $gs = 8$. This strategy is inefficient because the condition `temp` < T_0 is checked for each point output by the `GRID` operator.

The algebraic plan for selection is shown in Figure 3. Below, we walk through each of the operations used in this plan.

Variable Substitution. To improve on the gridding plan, the system needs to push the `GRID` operator up the plan past the filter operator, and execute the filter symbolically. Because all functions involved are polynomial, it is always possible to execute the filter using at least a partially symbolic approach. The first step is variable *substitution*, which replaces all the dependent variables in the tuple by their formulas, in each constraint in which they occur. In Figure 3, variable substitution replaces the linear constraint `temp` < T_0 with the polynomial $f(x, y) - T_0 < 0$. The result of substitution is a tuple in which all constraints, with the exception of functions, involve only independent variables and no dependent variables. If all the functions and constraints in the original tuple are polynomials, then the substitution result also contains only polynomials.

Solving/Approximation. The next step is to evaluate the filter symbolically. We use different approaches depending on the type of predicate. When there is a useful closed form solution to the filter constraints, we employ *equation solving*. For example, the 1D equation $f(x) > a$ (< and = are similar) where a is a constant can be solved for arbitrary-degree polynomials $f(x)$ (*e.g.*, using Newton-Raphson iteration) to yield a set of feasible intervals for x, which form bounding hypercubes for the output tuple(s) (Section 4.2).

For multivariate functions, closed form solutions are harder to compute. An equation like $f(x, y, z) = 0$ is "closed form", but does not necessarily help efficiently enumerate the grid points that satisfy this constraint, for arbitrary f. Our working definition of "closed form" is that it must be possible to transform the constrained region into a collection of hypercubes, which can then be gridded efficiently. For multivariate functions, it is not possible to represent an arbitrary region exactly as a set of hypercubes *e.g.*, in 2D, a circle cannot be decomposed exactly into a set of rectangles. Hence, for polynomial constraints, FunctionDB instead uses an approximation algorithm (Section 4.3) to decompose such regions into a set of hypercubes. The output of approximation is a maximal collection of hypercubes such that the grid points from these hypercubes all satisfy the original constraint(s), together with additional boundary points, which can be fed to a `GRID` operator (Figure 3). The boundary points are crucial, and help guarantee – despite our approximation – that we produce the same result tuples as the original plan where `GRID` was at the leaf.

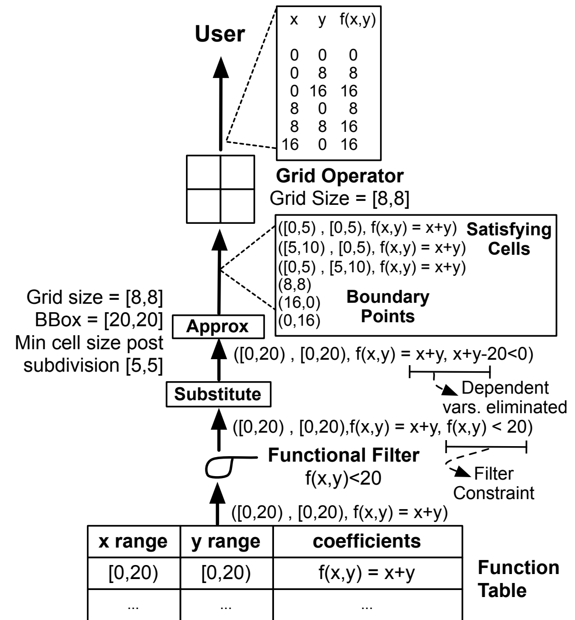

Figure 3: Algebraic plan for `SELECT x,y WHERE` $f(x, y) < 20$**.**

Joins. Executing joins between function tables is surprisingly similar to simple selections. The heuristic for plan generation, as with selections, is to push `GRID` to the top of the plan past the join operator. The join is executed by concatenating functions and constraints from each of the input tuples, and adding the join predicates as additional constraints. This is followed by operators for variable substitution and equation solving/approximation, as for selections. For example, suppose we have two models `temp` = $f_1(x, y)$ and `temp` = $f_2(x, y)$ stored in tables `model1` and `model2` respectively, and we want to find `x,y` locations where the model predictions differ by more than a threshold T_0. This is a join:

```
SELECT m1.x, m1.y FROM model1 as m1, model2 as m2
WHERE m1.x = m2.x AND m1.y = m2.y
AND |m1.temp - m2.temp| > T0
GRID m1.x gs, m1.y gs
```

The algebraic plan for this join is shown in Figure 4. The natural join on `x` and `y` merges the bounding rectangles of tuples coming from its left and right children, discarding pairs which are disjoint in either `x` or `y`, and creates a single bounding rectangle on `m1.x` and `m1.y` as independent variables. Natural join is a special op-

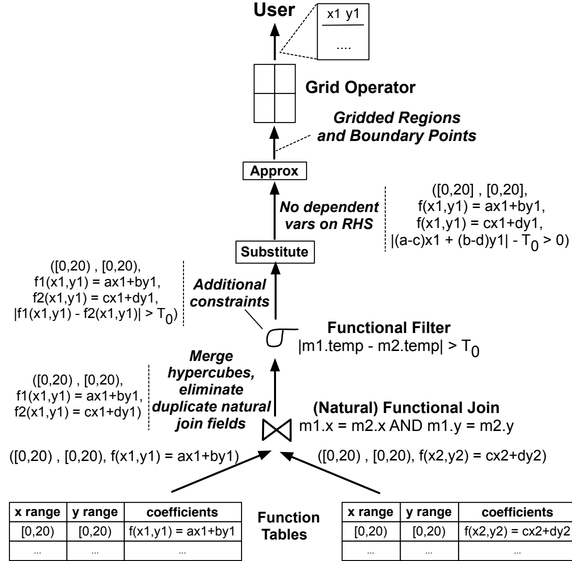

Figure 4: Algebraic plan for join to compare model predictions.

erator used here for convenience: instead of explicitly adding the constraints `m1.x = m2.x` and `m1.y = m2.y` and substituting them later, it combines these operations into one step and directly eliminates the duplicate `x` and `y` fields. We do need to add the constraint `|model1.temp - model2.temp|` $> T_0$ to each joined tuple. The tuples are fed to the substitution operator which substitutes for `model1.temp` and `model2.temp` in terms of `x` and `y`, followed by the approximation operator which approximates the resulting region, $|f_1(x,y) - f_2(x,y)| > T_0$. Finally, the `GRID` operator at the top grids each tuple output by the approximation operator, producing results conforming to grid semantics.

Joins between function tables and ordinary discrete tables are also useful *e.g.*, to look up model predictions at a set of discrete points. To execute such joins, we view each row of the discrete table as a constraint to add to each piece in the function table, and process the constraints algebraically, as usual. This approach may be inefficient for a large discrete table – we have not encountered this in our queries, but plan to address this in future work.

Function Inference. While substitution, equation solving and approximation suffice to support arbitrary select-project-join queries over polynomial functions, queries with equality predicates can be sped up further. The key idea is to transform equality constraints into functions, reducing the number of independent variables and hence the dimensionality of the space in which solving or approximation needs to be performed, greatly improving performance. Consider the `temp` $= f(x,y)$ example and suppose we want to find temperatures at locations on the line $x + y = 20$. This predicate can be transformed to a function: $y = 20 - x$, because $x + y$ is linear, and allows separating y as a function of x. The query is now much more efficient: we grid along only `x`, and for each grid point x, output $(x, y = 20 - x)$. More generally, define an expression $E(v_1, v_2, \ldots)$ to be *separable* on variable v_i if $E(v_1, v_2, \ldots) = 0$ can be transformed to an explicit equation for v_i in terms of the other variables: $v_i = E'(v_1, \ldots, v_{i-1}, v_{i+1}, \ldots)$, where E' is also a known type (*i.e.*, polynomial). Inference works for constraint expressions which are separable on one or more of their variable(s). Inference can also be performed on functions of a single variable, when they can be symbolically inverted *i.e.*, $y = f(x) \rightarrow x = f^{-1}(y)$. For the multivariate case, we only support separating linear constraints. We do not currently support inference for functions which are not single-valued and invertible, or for nonlinear constraints.

While inference is attractive from a performance viewpoint, one problem is that inference can change query semantics. If in the query above, the user specified grid sizes for both `x` and `y`, inference would violate grid semantics, because `y` values output by the inference plan would not necessarily be integer multiples of the grid size. Hence, we adopt a middle ground: we use variables in a query's `GRID` clause as hints to perform inference, and keep reducing a query's dimensionality until all and only the variables in the `GRID` clause are independent variables. For example, the query:

```
SELECT * WHERE x + y = 20 GRID x 0.5
```

would use inference (`x + y = 20` $\rightarrow$ `y = 20 - x`), but:

```
SELECT * WHERE x + y = 20 GRID x 0.5, y 0.5
```

would not. If it is not possible to achieve the user-specified basis using inference, the system throws a compile time error. For complex queries, where it is unreasonable to expect the user to choose the best basis, it would be interesting to infer the best possible basis automatically: we leave this to future work.

Substitution and inference help answer a very important class of application queries *without* requiring the user to specify a grid size: those that need to predict function values at some point using interpolation. For example, Query 1 (Section 2) looks up the temperature predicted by a model at a given location and time. This query involves three filter predicates: `x = 20`, `y = 20` and `time = 10`. After inferring constant functions from each of these constraints, and substituting them into the expression `temp = f(x,y,time)`, we get an equation of the form `temp` $= T_0$, which gives us a unique value for `temp`. Thus, the query has an *empty* basis following the inference step, and because the result is already finite, no further gridding is required to enumerate the answers.

Aggregate Queries. FunctionDB aggregate operators are the continuous analogues of traditional database aggregates like `SUM`, `AVG` and `COUNT`. Aggregates over discrete tuples generalize to definite integrals over continuous functions. For example, SQL `COUNT` applied to a relation R counts the number of tuples in R, essentially computing the sum $\Sigma_{v \in R} 1$. The continuous analogue is `VOLUME`, which measures the volume of the region defined by each tuple, summed over all its input tuples. This is the limit as the number of grid points goes to infinity of `COUNT` applied to a *gridded* version of each input tuple, because the number of grid points satisfying the constraints in each tuple is proportional to the generalized volume of the region (a 1D length, a 2D area or a volume in 3D and higher). The count converges to the integral $\int_R 1\, dV$, where R is the constrained region and dV is a volume element in that region. Other aggregates generalize similarly; see Table 1. In the table, R denotes a relation for the discrete operators, and the corresponding region for continuous operators. All formulas assume a single piece.

Aggregate operators live at the top of FunctionDB query plans. They can be performed symbolically provided the variable or expression being aggregated is a function which can be symbolically

SQL Aggregate	FunctionDB Analogue
COUNT $= \Sigma_{t \in R} 1$	VOLUME$(R) = \int_R 1\ dV$
COUNT$(R.x) = \Sigma_{t.x} 1$	VOLUME$(R.x) = \int_{R.x} 1\ d(R.x)$
SUM$(R.x) = \Sigma_{t \in R} t.x$	SUM$(R.x) = \int_R x\ dV$
AVG$(R.x) = \frac{\Sigma_{t \in R} t.x}{\Sigma_{t \in R} 1}$	AVG$(R.x) = \frac{\int_R x\ dV}{\int_R 1\ dV}$

Table 1: Discrete aggregates and FunctionDB counterparts.

integrated. We support this for polynomials, as well as some other classes of functions (Section 5.2 has an example). We compute the integral of each function over its hypercube of definition, and sum them (or for AVG, find the ratio of two sums). When symbolic integration is not feasible, we can either use pure gridding, or more sophisticated numerical integration to approximate the integral (we currently use gridding). Although there is no provable guarantee on the accuracy of aggregate results, we guarantee that if a region being aggregated needs to be approximated using hypercubes, a sampling size $\leq$ the grid size will be used to evaluate the aggregate. In practice, symbolic/partially symbolic aggregate plans are at least as (and typically more) accurate compared to gridding.

FunctionDB supports GROUP BY: we omit details here but explain how this works for one of the queries in our evaluation. Grouping can only be performed symbolically on independent variables – however, it is possible to group on a dependent variable if that variable can be transformed to an independent variable using inference.

Discussion and Subtleties. While grid semantics is well-defined and usually intuitive, it does not result in intuitive results for *zero measure* regions – those with zero volume in the space of independent variables *e.g.*, points in 1D, lines in 2D or areas in 3D. Such regions result from equality predicates — for example, `x + y = 25` is a line in the `x-y` plane. Gridding semantics applied to such regions produces irregular results or fails to produce *any* results whatsoever *e.g.*, none of the points on an `x-y` grid may lie exactly on `x + y = 25`. Here, we can use inference to transform this line to $y = 25 - x$, so this is not an issue. However, for predicates which cannot be transformed to functions (*e.g.*, nonlinear equations), we instead provide the user with an option to *widen* the query by relaxing the equality to an inequality – *e.g.*, this would replace `x + y = 25` with $25 - \epsilon < x + y < 25 + \epsilon$ using a user-specified ϵ. Second, aggregate semantics can also be non-intuitive over zero measure regions. Consider the query:

```
SELECT AVG(temp) FROM model WHERE x + y = 25
```

For a nonzero-measure region like `x + y < 25`, a valid implementation of this query is to enumerate grid points from the region and compute the average `temp` over these points. When the grid spacing approaches zero, the gridding average approaches the analytical average provided `temp` is a smooth function. However, somewhat counter-intuitively, this is *not* a valid way to compute the average `temp` over `x + y = 25` — the gridding average does not converge to the line integral of `temp` over this line. We currently solve this by enforcing widening (as explained above) for equality predicates, except when using inference. We hope to support line/surface integrals as first class primitives in FunctionDB in future work.

4. QUERY PROCESSING ALGORITHMS

We now describe the algorithms used for algebraic operations: substitution, inference, equation solving and approximation.

4.1 Substitution-Inference Loop

Filter and join operators in algebraic query plans are followed by operators that perform substitution and inference on the new constraint(s) introduced by the filter/join. While we have described substitution and inference as independent operations, in reality, they interact with each other and need to be performed together. Substitution introduces new constraints into a tuple, which can be potentially transformed to functions using inference. Conversely, inference adds new functions, and hence new dependent variables which must be substituted into other constraints. Hence, substitution and inference proceed in a *loop* until a fixed point is reached. This happens when two conditions are met: the set of independent variables is equal to the set of variables with GRID clauses, and dependent variables are not involved in any constraints apart from their defining functions. The first condition implies that no need for further inference, and the second that there is no need for further substitution. For example, the tuple $(x + y = 20, x + 2y < 30)$ would first be transformed to $(y = 20 - x, x + 2y < 30)$ using inference. Now substituting for y in the second constraint yields the final result: $(x > 10, y = 20 - x)$. Analyzing a query to determine substitutions and inferences to perform happens at compile time, and the sequence is replayed at run time on each tuple. This gives up some flexibility *e.g.*, an arbitrary degree-2 polynomial on x and y might be separable for particular values of coefficients *e.g.*, $x^2 - y = 0 \rightarrow y = x^2$, but because it is not separable on x/y in the general case, we cannot use inference. With a compile time approach, the analysis need not be repeated per-tuple, and the semantics are consistent for all tuples.

4.2 Equation Solving In One Dimension

For 1D functions, we use a Solve operator to transform a set of predicates into a set of satisfying 1D intervals. This operator follows the substitution-inference operator, and precedes the GRID operator at the top of a query plan. For a function of one variable $y = f(x)$, predicates of the form $x > a$ on the independent variable, $y > a$ on the dependent variable, or more generally, $E(x, y) > a$ where E is a polynomial function can all be solved directly ($<$, $=$ are similar) . Selections on x are trivial *e.g.*, applying the predicate $x \geq 3$ to $y = 2x$ defined over $[2, 4]$ yields a piece with the same function, $y = 2x$, now defined over $[3, 4]$. More generally, the predicate $E(x, y) > a$ is simplified by substitution to the form $F(x) > a$. This can be solved for polynomial $F(x)$ using any solver which can find the zeroes of a polynomial function (*e.g.*, Newton-Raphson). Because a continuous function has alternating signs in the intervals of x between its zeroes (assuming tangent points are counted as multiple zeroes), we can output alternate intervals satisfying the inequality. While we have called the Solve operator "exact", we recognize that equation solvers are not exact and do have error; we are limited by the accuracy of the solver.

4.3 Approximating Multivariate Regions

FunctionDB uses a recursive *subdivision* algorithm to approximate regions defined by multivariate polynomial constraints with a collection of hypercubes. Our algorithm is adapted from Taubin's computer graphics *rasterization* algorithm [17]. Taubin's original algorithm displays polynomial curves and surfaces of the form $f(x, y, z, \ldots) = 0$. We show how to extend to $<$ and $>$ constraints and arbitrary logical combinations of constraints, and also show how to *normalize* axes, which is essential to achieve good performance for constraint approximation. The key idea is to recursively subdivide space into hypercubes, and test whether the hypercubes

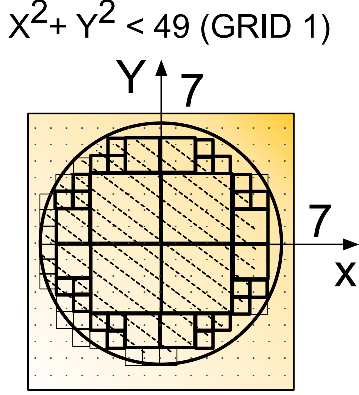

Figure 5: Subdivision to approximate a circular region.

lie within the constrained region. A hypercube entirely inside the constrained region can be added to the result list, and a hypercube which is entirely outside can be discarded. A hypercube which might contain points on the boundary of the region is subdivided further and tested. Figure 5 shows how subdivision works when the region being approximated is a circle. The key point to note is that on average, the approximating hypercubes are *much* larger than grid cells, and hence the algorithm outputs region representations an order of magnitude more compact than gridding – proportional to the boundary of the region, rather than its interior. Our experimental results in Section 5 (Table 2) confirm this to be the case in practice.

Without loss of generality, we assume a tuple containing a single polynomial constraint of the form $F(X) < 0$, where $X = \{X_1, X_2, \ldots\}$ is the set of independent variables, and a bounding hypercube over X (we later show how to extend to a tuple containing multiple constraints). The key step of the algorithm is testing whether a given hypercube H contains any point on the boundary of the constrained region *i.e.*, whether H contains a point X_0 such that $F(X_0) = 0$. If H contains no zeroes of $F(X)$, the continuity of $F(X)$ implies that all points in H have the same sign for $F(X)$, and it is sufficient to test the constraint at any corner of H to determine whether the entire hypercube passes or fails the constraint. However, if H might contain zeroes of $F(X)$, H needs to be subdivided further.

Taubin's Test. Taubin's test allows us to determine if an arbitrary polynomial has zeroes inside a square hypercube (*i.e.*, a hypercube whose sides are all equal). The key idea is to translate the polynomial $F(X)$ to a reference frame whose origin is at the *centre* of the hypercube. The advantage of this reference frame is that the absolute value of each of the variables is upper bounded by s, where s is half the side of the hypercube. Hence, it is possible to bound the absolute value of the polynomial in the new reference frame, and use this bound to decide if the polynomial can have a zero.

More formally, the test works as follows: consider a single term in the translated polynomial $F'(X)$ of the form: $C_{\alpha_1,\alpha_2,\ldots}X_1^{\alpha_1}X_2^{\alpha_2}\ldots$. If $d = \alpha_1 + \alpha_2 + \ldots$ is the degree of the term, the term is bounded in absolute value by $|s^d C_{\alpha_1,\alpha_2,\ldots}|$. If we use C_d to denote the sum of absolute values of the coefficients with degree d, then this implies that the sum S of all the non-constant terms in $F'(X)$ is bounded absolutely by the sum $S_{max} = \Sigma_{d=1}^{d_F}|C_d s^d|$, where d_F is the degree of F'. If we denote the constant term in $F'(X)$ by C, then $F'(X) = C + S$. Hence $|F'(X)| \geq |C| - |-S| \geq |C| - |S|$, which implies $F'(X)$ cannot have a zero in the hypercube if $|C| > |S|$, and hence if $|C| > S_{max}$, which is the final form of our test. This test is one-sided *i.e.*, if $|C| > S_{max}$, then this guarantees that the original polynomial $F(X)$ does not have any zeroes in the hypercube, but the converse is not true: $|C| \leq S_{max}$ does not imply that $F(X)$ must have zeroes in the hypercube. This is fine from a correctness standpoint, though a hypercube lying entirely inside or outside the region may be subdivided unnecessarily. In practice, the test works well on regions with up to 4 dimensions (Section 5), correctly discards most non-intersecting hypercubes, and is less expensive than gridding, whose cost is proportional to the volume/area of the region.

Stopping Criterion. A region like a circle cannot be exactly decomposed into bounding hypercubes, and hence subdivision will continue to yield squares that partially overlap the region, no matter how small the subdivided squares. Here, the grid size for the overall query is a good stopping criterion. Our algorithm stops subdividing a square when it is smaller than the grid size, because this is sufficient to achieve grid semantics. To actually ensure grid semantics, we modify the algorithm slightly: all hypercubes with side smaller than the grid size are themselves subject to the `GRID` operator and the resulting grid points are manually checked against the filters. The result of approximation is thus a collection of both hypercubes and boundary points, and it is easy to see that gridding this output yields the same result as gridding the original tuple.

Algorithm 1: Algorithm To Approximate $F(X) < 0$

Given: A constraint $F(X) < 0$, a bounding hypercube $H(X)$ over the vector X of independent variables, and grid sizes $g(X)$ for each of the independent variables.

1. $[H^n(X), F^n(X), g] \leftarrow$ Prenormalize($H(X)$, $F(X)$, $g(X)$)
2. SquareList $\leftarrow \{H^n(X)\}$
3. ResultList $\leftarrow \{\}$
4. **while** *SquareList has more tuples* **do**
5. $H_{next}(X) \leftarrow$ SquareList.Front()
6. $l \leftarrow H_{next}(X)$.side
7.   **if** $l < g$ **then**
8.   CandidatePoints $\leftarrow$ Grid(H_{next})
9.   BoundaryPoints $\leftarrow \{P \in CandidatePoints : F^n(P) < 0\}$
10.   ResultList.Add(BoundaryPoints)
11.   continue **while**
12.   **else**
13. $F'(X) \leftarrow$ Translate($F^n(X)$, Centre(H_{next}))
14.   **for** $i \leftarrow 0$ *to* $d_{F'}$ *(degree of* $F'(X)$*)* **do**
15. $S_i \leftarrow \Sigma|T_i|$, the degree i terms in F'
16.   **if** $S_0 - \Sigma_{i=0}^{d_{F'}}(S_i \times l^i) \leq 0$ **then**
17. $\{H_{small}\} \leftarrow$ Subdivide(H_{next})
18.   SquareList.Add($\{H_{small}\}$)
19.   **else**
20.   **if** *Corner of* H_{next} *satisfies* $F^n(X) < 0$ **then**
21.   ResultList.Add(H_{next})
22. Return Renormalize(ResultList)

The subdivision algorithm is summarized in Algorithm 1. For the "Translate" step, we use Horner's rule, an efficient form of polynomial long division. We now discuss some practical details not covered above or by Taubin's original test.

Prenormalization. The above stopping criterion does not support unequal grid sizes along different dimensions. A related drawback is that different variables in our models can have very different units or orders of magnitude *e.g.*, time and distance. For example, if the range of values for a time is $[0, 100]$ but that for a distance is $[0, 10000]$, then the bounding square would have to start with side $10,000$ along both dimensions, which is inefficient. To eliminate these problems, we added a prenormalization step where independent variables are scaled by the reciprocals of their respective grid sizes, and their coefficients in functions/constraints are scaled appropriately. For example, if the original tuple is $(0 \leq x < 10, 0 \leq y < 2, x^2 + y^2 < 10)$, and the grid sizes for x and y are 5 and 1 respectively, the tuple after normalization would be $(0 \leq u < 2, 0 \leq v < 2, 25u^2 + v^2 < 10)$, and we use a unified grid size of 1 ("g" in Algorithm 1) as stopping criterion.

Logical Expressions. The need to support arbitrary `WHERE` predicates implies that the test for zeroes needs to be extended to logical expressions with multiple predicates (*e.g.*, $(x^2 + y^2 < 25) \wedge (x + y < 10)$). We sketch a procedure to test if a conjunction (logical `AND` expression) of the form $C_1 \wedge C_2 \wedge \ldots C_N$ is satisfied by any points inside a hypercube H (the algorithm for logical `OR` is the identical mirror image of this algorithm). Define the *sign* of a point with respect to a constraint to be + if it satisfies the constraint, and - if not. Pick a corner of H and compute its sign with respect to each of the C_i, taken individually. We consider three cases depending on signs for each of the C_i:

1. If all the signs are +, and none of the C_i have zeros in the hypercube, then no C_i can change from + to -; hence the conjunction has sign + throughout H.
2. If any of the signs are -, for each such C_i, test if it has a zero in H. If there exists a C_i with no zero in H, it cannot change from - to +; hence the conjunction has sign - throughout H.
3. Otherwise, H needs to be subdivided further.

This algorithm works correctly when the test for zeroes is one-sided, which is important when using Taubin's test for individual C_i. While we have assumed C_i to be elementary, extending to when the C_i's themselves are logical expressions is straightforward.

5. EVALUATION

We experimentally evaluate FunctionDB on the queries described in Section 2. We quantify two advantages of our algebraic query processor over the pure gridding approach used by systems like MauveDB. First, a pure gridding approach requires the system to read a large number of discrete points and process them, resulting in poor performance in terms of both CPU and I/O cost. A symbolic approach only processes as many tuples as there are pieces in the function table, and similarly, our subdivision-based approximation approach performs less computation and I/O than gridding. Second, while widening grid spacing can result in better performance, a representation with a wide grid interval is insufficient to answer selection queries requiring finely gridded answers, and introduces significant *discretization error* in aggregate queries, because a wide grid is a coarse approximation to a continuous function. In contrast, symbolic algorithms do not introduce discretization error.

Because the performance of FunctionDB depends on how compact our model representations are, the segmentation algorithm and parameters we use impact our evaluation results. An algorithm which uses very few pieces to fit the data would mean a compact model that is extremely fast for query processing, but perhaps less accurate if this approximation is not valid. An algorithm which uses more pieces might be more accurate, but slower for query processing because it is less compact. On the other hand, an algorithm which is overly aggressive and creates too many pieces can, somewhat counter-intuitively, also result in reduced prediction accuracy, due to overfitting and reduced robustness to outliers. While we recognize that this accuracy-performance tradeoff is very important in practice, we do not focus on segmentation in this paper. We instead pick all our models to minimize *cross-validation error*. Cross-validation is a theoretically sound model fitting strategy which divides the available raw data into training and test sets in multiple ways, and picks model parameters to minimize the average error on the test sets. For a more detailed discussion of the accuracy-performance tradeoff resulting from segmentation strategies, we refer the interested reader to a related thesis [18] which explores this tradeoff experimentally for our applications.

Experimental Methodology. We have built a C++ prototype of FunctionDB. We constructed query plans by connecting operators by hand, following the plan generation rules in Section 3. For each experiment, we compare symbolic and pure gridding versions of the same query, and in addition we compare to PostgreSQL for some experiments. In all our experiments (excepting Postgres), the system reads data on disk into an in-memory table, executes the query in-memory, and writes results to disk. We built an in-memory prototype for simplicity because our data sets fit in main memory. Some experiments quantify I/O cost (reading data from disk) separately from CPU cost (query processing). CPU cost measures execution time when all data is already in the DBMS buffer pool, and I/O cost indicates the compression gain achieved by a compact symbolic representation of the data. We ran all experiments on a 3.2 GHz P4 single processor machine with 1 GB RAM and 512KB L2 cache. All results are averaged over 10 runs.

5.1 Temperature Application

We evaluated FunctionDB on the temperature data described in Section 2. This data contains $\approx$ 1,000,000 temperature observations from 54 sensors with schema `(x, y, time, temp)`. We fit two kinds of models to the data. The first is `timemodel`, a piecewise linear regression model for `temp` as a function of `time`, fit using a peak finding procedure as described earlier. We used cross validation to pick the knob for the peak finding strategy. This model contains 5360 functions, each piece fitting $\approx$ 200 raw temperature readings. The second is `3dmodel`, a piecewise quadratic model of the form `temp` = Q(`x`,`y`) + L(`time`), where Q is a degree 2 polynomial and L is a linear function. We fit this model using GUIDE, a popular regression tree package [13], which uses a tree pruning strategy to reduce generalization error. `3dmodel` contains only 22 pieces because it is fitted over a shorter time interval. We evaluated a variety of queries over both `timemodel` and `3dmodel`, which we detail below.

1D Histogram Query. We evaluated the histogram query (Query 4, Section 2) over `timemodel`. The query computes a histogram of temperatures over the time period of the data set, using temperature bins of width B_0 (the `GROUPSIZE` parameter). For each bin, the histogram height measures the total length of time (summed over all sensors) for which any of the sensors experiences a temperature in the range described by that bin. Because the `GROUP BY`

is over the dependent variable `temp`, we cannot directly do symbolic evaluation. The first step in the query plan is an inference operator, which transforms `temp = f(time)` to its inverse, `time` = f^{-1}`(temp)` and converts the basis to one where `temp` is the independent variable. It is now possible to group by `temp` by splitting or combining the bounding intervals for `temp` to assemble bins of size B_0. The topmost operator in the plan is a `VOLUME` aggregate applied to the now-dependent `time`. The aggregation is symbolic, and sums the lengths of all the time intervals in each temperature bin. The gridding approach reads a representation of the model gridded on `time` off disk and processes it. This plan uses a traditional `GROUP BY` to map `temp` values to bins of size B_0, and counts the time samples in each bin.

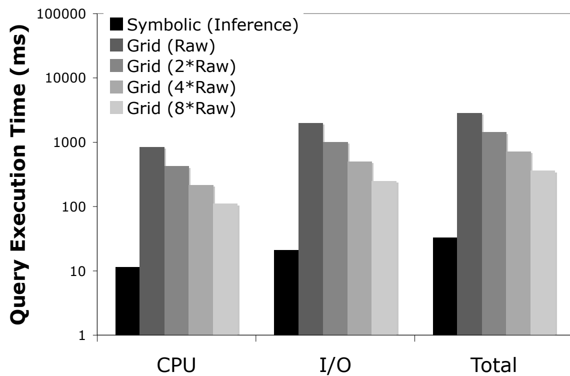

Figure 6: 1D Histogram Query: Performance Comparison.

Figure 6 compares the performance of symbolic and gridding approaches, for 4 values of grid size: 1 sec (the spacing of the original raw data), and 2, 4 and 8 secs (beyond this, the loss of accuracy due to gridding is too great). The symbolic plan is faster than the gridding strategies by an order of magnitude in terms of both CPU and I/O cost. Both savings are due to the small footprint of symbolic execution: the query plan needs to read, allocate memory for and process only 5360 tuples (1 per function). On the other hand, "Grid (Raw)" — gridding at the same spacing as raw data, for example, needs to read and process ~ 1,000,000 discrete points, and hence performs an order of magnitude worse.

Figure 6 shows that it is possible to reduce gridding footprint and processing time, by widening grid size. However, this adversely impacts query accuracy. Figure 7 shows the discretization error introduced by gridding as a function of grid size, averaged over all bins. The error is computed as percent deviation from the symbolic result, which does not discretize the data. Gridding suffers this error because it samples temperature at discrete time intervals and counts the number of observations in each bin. Sampling is inaccurate when the sampling interval is comparable to the time scale over which temperature varies significantly. As the figure shows, the error is higher when grouping temperature into smaller bins, and also grows with grid size. Hence, using a widely spaced grid is not a viable option. We note that while symbolic results are used as the baseline for computing error, these results are also limited by inherent error in the raw data and the model. We only intend to show that gridding introduces significant *additional* error.

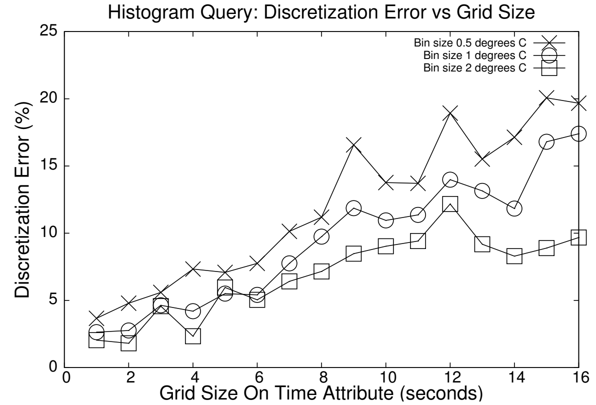

Figure 7: 1D Histogram Query: Error due to Gridding.

2D Selection Query. We next consider Query 2 from Section 2, which finds locations where temperature drops below a threshold:

```
SELECT x,y WHERE temp(x,y) < T0 GRID x gs, y gs
```

We ran this query over a slice of our 3-dimensional model, `3dmodel` at a particular instant of time. The symbolic plan (Figure 3) uses subdivision to approximate the region `temp(x,y) <` T_0, while the gridding plan (Figure 2) manually checks the condition for each grid point. We picked different values of T_0 to vary the query selectivity (lower T_0, lower the selectivity, and more selective the query). Because this query would benefit from an index on `temp` when using the gridding approach, to get an idea of how symbolic execution compares to an index, we also ran the same query against a gridded version of the table in PostgreSQL, using a B+ Tree index on `temp`. Because the symbolic plan returns results conforming to grid semantics, all three queries produce identical results. Figures 8 and 9 show the performance results (CPU + I/O time) for two grid sizes, one narrow (0.1m) and the other as wide as possible (1m), comparable to the separation of individual pieces. Approximation outperforms simple gridding significantly for selective queries, and is roughly on par with the optimized Postgres plan using indices (for the wide plan, Postgres had too much overhead, so we did not compare to it). For highly selective queries, approximation also outperforms an index-based strategy (*e.g.*, Postgres) because it has a smaller memory and I/O footprint than a B-Tree over gridded data. Our results suggest that it is sometimes worthwhile to use gridding, in conjunction with indices, for not-so-selective queries: incorporating this optimization would be interesting future work.

2D Aggregate Query. While the benefits of symbolic execution are not very pronounced for simple selections, aggregate queries (which are highly selective) benefit greatly from symbolic execution. Suppose we extend the previous query to compute the area of the region(s) colder than T_0:

```
SELECT VOLUME(x,y) WHERE temp(x,y) < T0
GRID x gs, y gs
```

This is identical to the previous query, except that the symbolic plan adds a `VOLUME` operator at the top and the gridding plan adds a `COUNT` operator (scaled appropriately) to compute area. Figure 10 compares the performance of approximation, pure gridding and Postgres (with B+ Tree) on this aggregate query for $T_0 = 1^oC$ (highly selective). For narrow grid sizes, approximation wins by an order of

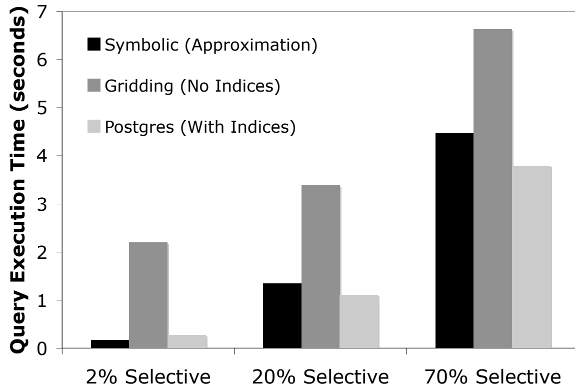

Figure 8: 2D Selection: Narrow Grid.

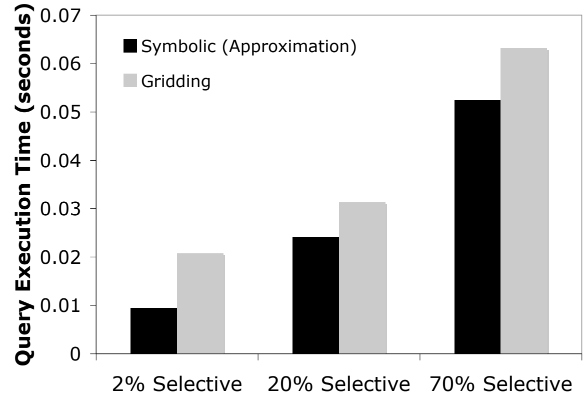

Figure 9: 2D Selection: Wide Grid.

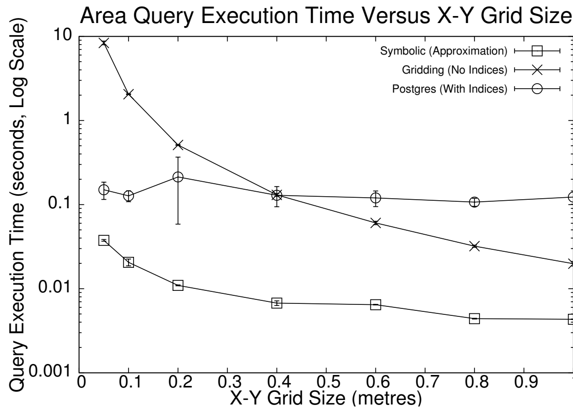

Figure 10: 2D Aggregate Query: Performance.

Grid Size (^oC)	0.1	0.2	0.4	0.8
# Hypercubes	497	249	113	50
# Grid Pts (No Indices)	1365001	341251	85351	21054
# Grid Pts (With Indices)	5478	1411	343	77

Table 2: Efficacy Of Approximation for Aggregate Query.

magnitude over both pure gridding and the index-based plan. For wide grid sizes, approximation still wins, but by a smaller factor. Accuracies were roughly comparable for both approaches, though more erratic for gridding with wide grid sizes. Table 2 compares three metrics for this query at different grid sizes: the number of hypercubes used to approximate the region of interest, the number of grid points evaluated if using gridding without indices, and the number of grid points evaluated if using an index on gridded data. We see that approximation produces more compact representations than gridding without indices by an order of magnitude, and still smaller than when using indices, explaining our performance wins.

3D Join Query. To understand how the approximation approach scales to higher dimensions, we evaluated Query 3 in Section 2, which is a self-join between two 3D function tables to find fluctuations in temperature between two given extremes T_1 and T_2, and times when these extremes occur. This involves a natural join on `x` and `y`, and has 4 independent variables (`x, y, m1.time, m2.time`) after the join. However, we can use inference to reduce the number of dimensions to 2. To see why, observe that the predicate `m1.temp =` T_1 becomes `Q(x,y) + L(m1.time) =` T_1 after substitution, which is separable on `time` because the degree of `time` in the LHS expression is 1. Hence, inference can transform this equation to `m1.time = Q'(x,y)`, and eliminate `m1.time` from the set of independent variables. The same goes for `m2.time`, and the result is a tuple where `x` and `y` are the only independent variables, and the `ABS` constraint on time difference reduces to a polynomial constraint in `x` and `y`. Converting `m1.time`, `m2.time` to dependent variables also introduces 4 new constraints, because we have to substitute for them in the original hypercube bounds on `m1.time` and `m2.time`. The substitution-inference operator in the symbolic plan is followed by approximation, which runs the subdivision algorithm over the logical AND of the above 5 constraints. The top of the plan is a `GRID` operator, as usual.

Because the gridded table is quite large and does not fit in memory for narrow grid sizes, we compare our symbolic plan to running the query in Postgres on the gridded data. Postgres uses a sort-merge join on `x` and `y` (because the gridded data is partially sorted) and a B+ Tree on `m1.temp` and `m2.temp`. To quantify the benefits of inference and approximation separately, we also compare to a (different) symbolic plan for the same query which does not use inference, and therefore runs the approximation algorithm in a 4D space: `m1.x, m1.y, m1.time, m2.time`. Because these two approaches do not use inference, as per the discussion in Section 3, we were forced to widen the exact equality criteria on `temp` by $\epsilon = 0.1^oC$ (on both sides), to get meaningful results.

The overall selectivity of this join is determined by T_1 and T_2. In our experiment, we fixed T_1 near the median temperature (17^oC) and ran experiments for $T_2 = T_1 + 1.5$ (higher selectivity, so less selective), as well as $T_2 = T_1 + 4$ (lower selectivity, so more selective). The Postgres and approximation-only versions used a grid size of 5 sec on time. Figure 11 shows the performance for these different selectivities, as well as for two different grid sizes on `x` and `y`, (varying the grid size on `time` yielded similar results) for the three strategies. Inference is a substantial win in all situations, by

an order of magnitude, because it reduces the overall dimensionality of the query. The results for the approximation-based version indicate how the performance of FunctionDB degrades for queries where inference is impossible and/or the number of dimensions is high: echoing our previous results, approximation performs much better (10x) on selective queries, but a less selective query at a wide grid spacing might be better off using gridding with indices. We note here that inference also wins in accuracy over gridding. The widening factor ϵ results in false positives, because gridding outputs some `x,y` locations satisfying the expanded criterion, but not the original criterion. For $\epsilon = 0.1$, we found 179 false positives.

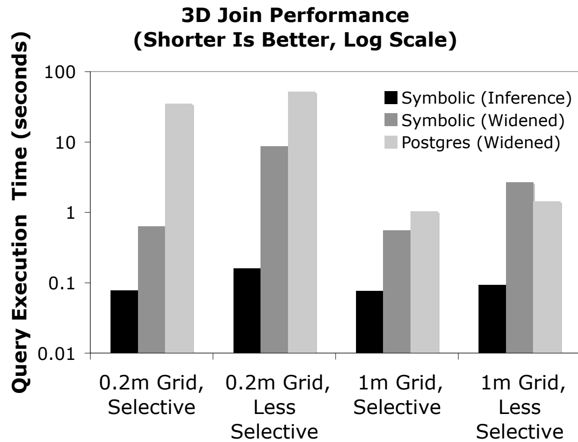

Figure 11: 3D Join: Performance.

5.2 Trajectory Similarity

Our second data set consists of 1,974 vehicle trajectories from Cartel, fitted with a piecewise 1D linear model for road segments. Our model has 72,348 pieces fit to 1.65 million GPS readings, and was validated using cross validation: our model had low cross-validation error ($\leq 10m$) for up to 20-30 missing GPS observations. For evaluation, we used the trajectory similarity query (Query 6) described in Section 2. Given a trajectory, the query finds neighbouring trajectories whose endpoints are within 1' of latitude/longitude ($\approx$ 1.4km). For each neighbour, the query lines up points on the given trajectory with points on its neighbour based on the "distance fraction" criterion explained earlier, and computes the average distance between pairs of points as a similarity metric.

To focus on join performance, we simplified the query by precomputing a materialized view `fracview` with the schema `(tid, lon, lat, frac)`. In the gridded model, points are sampled at regular intervals of the independent variable `lon`. For each point, `frac` $\in [0, 1]$ is the fraction of distance along the trajectory at which the point occurs. In the function table, `lat` and `frac` are both piecewise linear functions of `lon` (`frac` is linear because `lat` is linear, and the slope along each piece is constant). Given this, trajectory similarity becomes an equijoin on `frac`. The query plan is completely symbolic, the major step being the join, followed by inference to transform `frac` to an independent variable, and then `GROUP BY` over the `tid` field. The last step maps the expression for Euclidean distance to the functions in each tuple, and applies `AVG` to this expression. Although the square root results in a hyperbolic (non-polynomial) function, `AVG` can be computed symbolically because this function can be integrated symbolically.

The gridded version matches each grid point on the search trajectory with all points on its neighbour which lie within a widening criterion ϵ (we picked this to be twice the grid size). The average distance is computed between pairs of points lined up this way. Figure 12 shows the performance results, averaged over query trajectories. FunctionDB performs better than all but one of the grid sizes, and has no discretization error. The improvements are not as dramatic as with the temperature data, because the ratio of number of grid points to pieces is somewhat lower – but symbolic processing still performs significantly better than gridding. Gridding also introduces error because of the expanded criterion based on ϵ: we found that error could be as large as 15-30%.

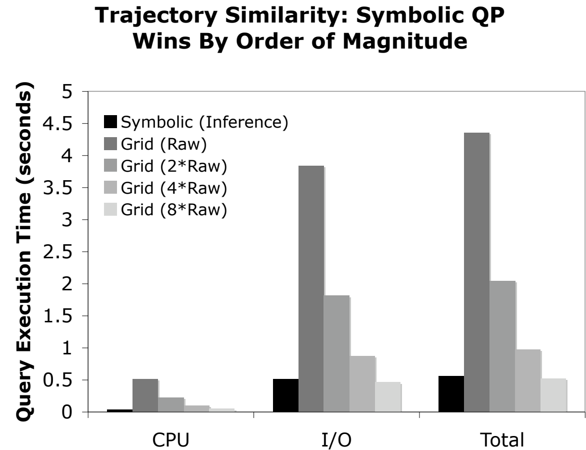

Figure 12: Trajectory Similarity: Performance Comparison.

6. RELATED WORK

Existing commercial DBMSs provide some support for models in the form of modeling tools for data mining *e.g.*, IBM's Intelligent Miner and Oracle Data Miner support PMML (Predictive Model Markup Language). However, these tools do not export a relational interface to model data, and instead view models as standalone black boxes with specialized interfaces for fitting. A typical use of PMML is to fit a set of points to functions using an external tool, load the functions into the DBMS, and then use the functions to predict the value of some other points. This is very different from our approach, where the functions themselves can be joined, aggregated, or queried. Systems such as Postgres with PostGIS [1] extensions and moving object databases [7, 19], support polyline and trajectory data as ADTs. Internally, these types have similar structure to regression curves, but they do not support the range of operations over functions which FunctionDB supports, and are targeted exclusively at geospatial/moving object data. GIS extensions cannot compute the average value of a curve over some range, join curves together, or convert to/from curves and discrete points.

MauveDB [4] also proposes querying models using a relational framework. FunctionDB uses an algebraic framework with well-defined semantics, which enables substantially faster and more accurate query execution than pure gridding as used by MauveDB. Pulse [2] is a continuous stream query processor that can fit one-dimensional polynomials (in time) to input streams. Query processing in Pulse is accomplished by finding algebraic solutions to systems of equations. Pulse, however, cannot work with functions of more than one variable or support arbitrary `WHERE` clause pred-

icates, both of which require adoption of a constraint-based model and approximate solutions, as we have discussed. Also, Pulse only supports a subset of the operations that FunctionDB does — *e.g.*, it only allows natural joins between two streams on time (with optional filters on dependent variables). Pulse does allow users to specify error bounds on query results, and back-propagates those bounds to drive the approximation granularity used to produce discrete results. Because it is important for users to have an idea of end-to-end query error whenever any form of modeling is used, this would be an interesting and useful feature to add to FunctionDB.

Representing infinite relations, and query processing using algebraic computation are not new ideas. Constraint query languages [10] represent and query infinite regions defined by systems of constraints. There have been prototypes of constraint databases for solving spatial queries [3, 5, 6, 15, 16], using some techniques similar to ours (*e.g.*, equation solving, substitution). FunctionDB differs from constraint databases significantly, because we do not aim to find closed form solutions for arbitrary constraints. Hence, we can leverage approximation to support a wide class of polynomials, whereas constraint databases tend to focus on linear constraints. Non-linear constraint solvers with algorithms similar in spirit to ours do exist [8], but they aim for closed form solutions, and do not support relational/aggregate operations.

We note that Taubin's test is a special instance of interval arithmetic methods [14]. We could use any of these algorithms in place of Taubin's. [14] indicates Taubin's test to be among the best performing of these algorithms.

7. CONCLUSION

We described FunctionDB, a novel DBMS that supports regression functions as a data type that can be queried and manipulated like traditional relations using an algebraic query processor. We have shown that our query processor is 10x-100x faster, as well as 15-30% more accurate on several real queries, compared to existing approaches that represent models as gridded data. For less selective queries, we do find that gridding occasionally performs better when used in conjunction with indices, and we hope to use our results as a guide to building a query optimizer for FunctionDB in future.

8. REPEATABILITY ASSESSMENT

The results in this paper, except Table 2 (added after our submission), were verified by the SIGMOD repeatability committee.

9. ACKNOWLEDGEMENTS

This paper was supported by the NSF under grant number 0448124. We thank the anonymous reviewers for their insightful comments and valuable feedback on the paper, and the SIGMOD repeatability committee for taking the time to repeat and verify the experimental results in this paper. We thank Adam Marcus, Ramki Gummadi, Yang Zhang and Vladimir Bychkovsky for reading through drafts of this paper and offering valuable suggestions.

10. REFERENCES

[1] PostGIS. `http://postgis.refractions.net/`.

[2] Y. Ahmad and U. Çetintemel. Declarative temporal data models for sensor-driven query processing. In *DMSN*, 2007.

[3] A. Brodsky, V. E. Segal, J. Chen, and P. A. Exarkhopoulo. The CCUBE Constraint Object-Oriented Database System. In *SIGMOD*, 1999.

[4] A. Deshpande and S. Madden. MauveDB: Supporting Model-Based User Views in Database Systems. In *SIGMOD*, 2006.

[5] S. Grumbach, P. Rigaux, and L. Segoufin. The DEDALE system for complex spatial queries. In *SIGMOD*, pages 213–224, 1998.

[6] S. Grumbach, P. Rigaux, and L. Segoufin. Manipulating Interpolated Data is Easier than You Thought. In *The VLDB Journal*, pages 156–165, 2000.

[7] R. H. Guting, M. H. Bohlen, M. Erwig, C. S. Jensen, N. A. Lorentzos, M. Schneider, and M. Vazirgiannis. A Foundation for Representing and Querying Moving Objects. *ACM TODS*, 25(1):1–42, 2000.

[8] D. Haroud and B. Faltings. Global consistency for continuous constraints. In *Principles and Practice of Constraint Programming*, pages 40–50, 1994.

[9] B. Hull, V. Bychkovsky, Y. Zhang, K. Chen, M. Goraczko, A. K. Miu, E. Shih, H. Balakrishnan, and S. Madden. CarTel: A Distributed Mobile Sensor Computing System. In *Sensys*, Boulder, CO, November 2006.

[10] P. C. Kanellakis, G. M. Kuper, and P. Z. Revesz. Constraint Query Languages. In *PODS*, 1990.

[11] E. J. Keogh, S. Chu, D. Hart, and M. J. Pazzani. An Online Algorithm For Segmenting Time Series. In *ICDM*, pages 289–296, 2001.

[12] R. A. O. L. Breiman, J. H. Friedman and C. J. Stone. *Classification And Regression Trees*. Wadsworth International Group, 1984.

[13] W. Y. Loh. Regression Trees With Unbiased Variable Selection And Interaction Detection. *Statistica Sinica*, 12:361–386, 2002.

[14] R. Martin, H. Shou, I. Voiculescu, A. Bowyer, and G. Wang. Comparison of Interval Methods For Plotting Algebraic Curves. *Computer Aided Geometric Design*, 19(7):553–587, 2002.

[15] P. Z. Revesz. Constraint databases: A survey. In *Semantics in Databases*, pages 209–246, 1995.

[16] P. Z. Revesz, R. Chen, P. Kanjamala, Y. Li, Y. Liu, and Y. Wang. The MLPQ/GIS Constraint Database System. In *SIGMOD*, 2000.

[17] G. Taubin. Rasterizing algebraic curves and surfaces. *IEEE Comp. Graphics and Applications*, 14(2):14–23, 1994.

[18] A. Thiagarajan. Representing and Querying Regression Models in an RDBMS. Master's thesis, MIT, Sep 2007.

[19] M. Vazirgiannis and O. Wolfson. A Spatiotemporal Model and Language for Moving Objects on Road Networks. In *SSTD*, pages 20–35, 2001.