Representing and Querying Regression Models in a Relational Database Management System

by

Arvind Thiagarajan

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

August 2007
[September 2007]

© Massachusetts Institute of Technology 2007. All rights reserved.

Author ....
Department of Electrical Engineering and Computer Science
August 10, 2007

Certified by ....
Samuel Madden
Associate Professor, MIT EECS
Thesis Supervisor

Certified by ....
Hari Balakrishnan
Professor, MIT EECS
sor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
OCT 1 2 2007
LIBRARIES

BARKER

Representing and Querying Regression Models in a Relational Database Management System

by

Arvind Thiagarajan

Submitted to the Department of Electrical Engineering and Computer Science
on August 10, 2007, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

Curve fitting is a widely employed, useful modeling tool in several financial, scientific, engineering and data mining applications, and in applications like sensor networks that need to tolerate missing or noisy data. These applications need to both fit functions to their data using regression, and pose relational-style queries over regression models.

Unfortunately, existing DBMSs are ill suited for this task because they do not include support for creating, representing and querying functional data, short of brute-force discretization of functions into a collection of tuples. This thesis describes FunctionDB, a novel DBMS that extends the state of the art. FunctionDB treats functions output by regression as first-class citizens that can be queried declaratively and manipulated like traditional database relations. The key contributions of FunctionDB are a compact, algebraic representation for regression models as piecewise functions, and an algebraic query processor that executes declarative queries directly on this representation as combinations of algebraic operations like function inversion, zero finding and symbolic integration.

FunctionDB is evaluated on two real world data sets: measurements from a temperature sensor network, and traffic traces from cars driving on Boston roads. The results show that operating in the functional domain has substantial accuracy advantages (over 15% for some queries) and order of magnitude (10x-100x) performance gains over existing approaches that represent models as discrete collections of points. The thesis also describes an algorithm to maintain regression models online, as new raw data is inserted into the system. The algorithm supports a sustained insertion rate of the order of a million records per second, while generating models no less compact than a clairvoyant (offline) strategy.

Thesis Supervisor: Samuel Madden
Title: Associate Professor, MIT EECS

Thesis Supervisor: Hari Balakrishnan
Title: Professor, MIT EECS

Acknowledgments

I would like to thank my advisors, Sam Madden and Hari Balakrishnan, for introducing me to research and helping me sample and find interesting problems to work on.

It was a great experience working with Sam on FunctionDB: he was always willing to spend time and meet with me weekly, and this thesis would not have been possible without his advice, feedback, encouragement and constant help with bouncing ideas and directions off him.

I am also very grateful to Hari, who has always been exciting and lively to work with, thanks to his knack for spotting problems/issues early on, and never failing to come up with challenging questions which cut to the heart of the problem we were working on. Hari's comments and feedback after a group meeting talk I gave were particularly helpful to polish my work and round off this thesis.

I would like to thank Mike Ahern, who worked with me on the original class project idea that ultimately became this thesis, and contributed a lot by way of ideas, design discussions and feedback.

I would like to thank Vlad, Bret, Yang and Jakob from the Cartel project who were very helpful with sharing their datasets, which contributed substantially to the experiments in this thesis, as well as their expertise.

I also thank all my office mates at MIT – Mike, Mythili, Jakob and Emil, as well as many friends in G9 and CSAIL (too numerous to name) with whom I've had many stimulating conversations and discussions, and who helped make MIT a really great place to work at.

I thank my parents and grandparents for their unconditional support and encouragement at all times.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Relational databases have traditionally taken the view that the data they store is a set of discrete observations. This is clearly 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. For researchers in many fields, from social sciences to biology to finance [15–17], a common first step in understanding a set of data points is to model those points as a collection of curves, typically generated using some form of *regression* (curve fitting). Regression, a form of modeling, helps smooth over errors and gaps in raw data points, yields a compact and more accurate representation of those points as a few parameters, and provides insight into the data by revealing trends and outliers.

Many of the applications mentioned above, in addition to fitting models, need to ask relational-style queries *after* regression has been applied to their input data. For example, a sensor network monitoring an environmental variable like temperature or pressure may need to know when the variable crosses a threshold (a *selection* query), when the variable differs sharply between nearby locations (a *join* query), or the value of the variable averaged over time windows or geographic regions (an *aggregate* query). However, in the context of these applications, it is not desirable or feasible to directly query raw data. This is because raw data are either missing, noisy or simply unavailable in many situations. For example, in the sensor network application, it may be necessary to interpolate or extrapolate the data to predict sensor readings at locations where sensors are not physically deployed. Also, cheap sensors can occasionally fail or malfunction, or report garbage values due to low batteries and other anomalies. These outliers or garbage values need to be eliminated or smoothed by modeling.

Mathematical and scientific packages like MATLAB, Mathematica, Octave and R [1–4] do support creating regression models. However, these tools lack support for declarative or relational queries. Queries typically need to be implemented as custom scripts in MATLAB, or in languages like Perl. A related concern is that tools like MATLAB do not provide a seamless way to interact with data already stored in a DBMS. Data from a relational table needs to be manually imported into MATLAB in order to fit a regression model to it. Once a model has been fit to the data, it can be used to make predictions or compute the interpolated value of a function at specific points from within MATLAB, but this code lives in custom scripts, which do not provide any of the benefits of storing the data within a modern database system – for example, declarative queries, indexability, optimizability, transactions and recoverability, and integration with other data in the DBMS.

In order to solve this problem, this thesis proposes to push first-class support for regression models *into* a relational DBMS. This thesis presents FunctionDB, a system that allows users to directly query the functions output by regression inside a database system. By pushing this support into the database, rather than requiring the use of an external curve fitting and analysis tool, users can manage and query these models declaratively just like any other data, providing the benefits of declarative queries, indexability, and integration with existing database data.

Thus, FunctionDB is designed to help users who need to use regression models to manage their data, particularly when the raw data is noisy or has missing values. FunctionDB is essentially a relational database system with support for special tables that can contain functions, in addition to standard tables with raw (discrete) data. In addition to managing raw data, FunctionDB provides users with tools to fit that raw data with one or more functions. For example, FunctionDB might represent the points $(t = 1, x = 5), (t = 2, x = 7), (t = 3, x = 9)$ as the function $x(t) = 2t + 3$.

Once in this curve domain, it is natural to ask questions over the fit data directly, looking, for example, for curves that intersect, are confined within a certain area, or that have the maximum slope. Accordingly, FunctionDB allows users to pose familiar relational and declarative queries (*e.g.*, filters, joins, maps and aggregates) directly over functions. To this end, FunctionDB includes a novel *algebraic* query processor that executes relational queries on such functions by using direct symbolic algebra (*e.g.*, solving symbolic equations, symbolic integration and symbolic differentiation).

This thesis shows how relational and declarative operations traditionally supported by DBMSs are translated to algebraic manipulations in this functional domain. As a simple example, a selection query that finds the time when the temperature of a sensor whose value is described by the equation $x(t) = 2t + 3$ crosses the line $x = 5$ requires solving the linear equation $5 = 2t + 3$ to find $t = 1$. Similarly, the symbolic analogs for aggregate queries and join queries, while more complex than simple filters, can be described in terms of integration and function inversion respectively.

The most closely related work to this thesis is the MauveDB system [7], which also proposes to integrate models directly into a database system, but performs query processing over those models by storing them as gridded data points that can be fed directly into existing relational operators. For example, a curve like $y(x) = 2x + 1$ would be represented as a set of discrete data points in MauveDB – *e.g.*, $(0, 1), (1, 3), (2, 5) \ldots$, which can then be queried. In contrast, though FunctionDB also supports gridding to expose answers to users in the traditional format, the actual query processing is done directly over functions (and hence ungridded data), which yields order of magnitude efficiency gains, and substantial accuracy gains over the MauveDB approach, as this thesis demonstrates.

FunctionDB is also related to constraint databases and constraint query languages [20, 21], which were first proposed in the context of spatial database systems, and allow users to pose queries over systems of algebraic constraints. In contrast to constraint databases, FunctionDB views query processing in terms of algebraic primitives applied to functions, rather than in terms of solving systems of constraints. As this thesis shows, this restriction enables a simpler design and data model that are more extensible to a variety of regression functions.

1.1 Contributions

- We propose a compact representation for regression models as collections of piecewise functions, and a novel algebraic query processor that operates directly on this representation without first "materializing" functions into discrete data points.

- We show how to describe relational query processing operators in terms of algebraic primitives applied to functions, including function evaluation, zero finding, inversion and symbolic integration. The framework and algorithms proposed in this thesis are confined to functions of a single variable, but the system gracefully degrades to execute queries approximately (with well defined semantics) on functions of more than one variable, and on functions for which symbolic query evaluation is not feasible.

- For the case of one-dimensional time series data, we propose and evaluate an algorithm for online model maintenance as new raw data is inserted into the system. The algorithm supports leverages batch updates to avoid rerunning the regression fit for each new record inserted into FunctionDB.

- We evaluate the query processing and update algorithms used in FunctionDB using two real world data sets: a collection of temperature data from 54 temperature sensors in a building deployment, and a collection of GPS traffic traces collected from cars driving on Boston roads, from the Cartel [8] project at MIT. We find that FunctionDB achieves order of magnitude (by a factor of 10x-100x) better performance for aggregate queries and substantial savings (by a factor of 2x-4x) for other selective queries, when compared to approaches that represent and process regression models as discrete points. Also, we show that modeling raw data using FunctionDB can help interpolate a significant amount of missing data with high accuracy (up to 1 minute of GPS readings in the traffic trace experiment), thus providing more accurate results than query processing over raw data. Finally, we demonstrate that batching insertions enables FunctionDB to support a sustained (in-memory) insertion throughput of the order of a million records per second, which should be sufficient for many applications of interest.

Chapter 2

Example Applications and Queries

This chapter introduces and motivates FunctionDB through two applications: an indoor sensor network and an application that analyzes data from car trajectories. Queries for both applications are illustrated in the FunctionDB query language, which is essentially standard SQL with extensions to support regression. Real data and queries from these applications are also used in the evaluation (presented in Chapter 4).

2.1 Indoor Sensor Network

2.1.1 Application Scenario

In this application, a network of temperature sensors are placed on the floor of a building. Each sensor produces a time series of temperature observations. We use real data from an indoor sensor deployment at Intel Research, Berkeley that monitors several environmental variables, including temperature and humidity.

Below, we consider several queries that users of this application might want to ask:

1. What is the temperature at a particular sensor location at a given time?

2. At what times is the heating insufficient? For example, does the temperature drop below 18^oC anywhere, and when?

3. Compute the average of temperature at a particular location over a specified time window (*i.e.*, between times `t1` and `t2`).

4. Over a longer time scale (like a day) what does the histogram of temperature with time look like at a particular location? Equivalently, retrieve a histogram of temperatures ordered by the duration of time for which the location experiences that temperature. This might help detect locations prone to getting hot at particular times of the day when they face the sun.

Using a regression model to fit the temperature data is useful in this application for two reasons. First, radios on sensor nodes lose packets (sometimes as high as 15-20%). Second, nodes themselves fail (*e.g.*, when batteries are low), producing no readings or even garbage data. Hence, this application needs to cope with missing, as well as incorrect and noisy data.

While interpolation can be accomplished by simple averaging, regression is a more robust alternative and provides more insight into the data. Regression takes as input a dataset with two or more correlated variables (like time and temperature), and produces as output a formula for one or more of the variables, termed the *dependent* variables, as a function of the other variables, termed the *independent* variables. The aim of regression is to produce a model that approximates the ground truth data with as little error as possible. In cases where the raw data is known to be accurate and is hence the ground truth, regression serves as a compact approximation to the raw data, as well as helping interpolate gaps in the data. In other situations, such as the temperature application considered in this section, where the raw data itself is noisy due to measurement errors and/or outliers, a scientifically motivated regression model (constructed with prior knowledge of trends in the dataset) can help smooth over measurement error.

The simplest and most common algorithm used to fit regression models to raw data is linear regression. Linear regression takes a set of basis functions of the independent variable(s) *e.g.*, x, x^2, x^3 and computes coefficients for each of the basis functions (say a,b,c) such that the sum of the products of the basis functions and their coefficients ($ax + bx^2+cx^3$) produces a minimum-error 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 used.

As an example, a linear regression function might approximate the raw data points (x = 0, y = 0.1), (x = 1, y = 0.9), (x = 2, y = 2.2), (x = 3, y = 2.9), (x = 4, y = 4), (x = 5, y = 4.9) with the line $y = x + 1$.

In the temperature application example, a regression model for temperature (the dependent variable) as a function of time (independent variable) is preferable to simple averaging, for two reasons. First, regression models the underlying phenomenon, which is a continuous function of time. Interpolation and extrapolation both fit naturally into this framework. Second, when there is sufficient data, regression is less prone to outliers compared to local averaging.

A standard way to use regression when modeling data is to first *segment* data into regions within which it exhibits regularity or a well-defined trend, and then choose basis functions that are most appropriate to fit the data in each region. We term this process *piecewise regression.*

Figure 2-1 illustrates piecewise linear regression for temperature data from one of the sensors in the Intel Lab deployment. The plot shows a snapshot of temperature data from one of the sensors in the Intel Lab dataset, and a piecewise linear regression model that has been fit to the observed data. Each linear piece is defined over a time interval, and captures an increasing or decreasing trend in temperature over that interval (the temperature increases during the morning and decreases later in the day).

2.1.2 FunctionDB View Definition

This section shows how creating a regression view of temperature data can be accomplished with FunctionDB, and illustrates how each of the queries mentioned in the previous section would be expressed in the FunctionDB query language. We assume that raw temperatures are stored in a standard relational table, `tempdata` and the schema looks like `<ID,x,y,time,temp>` – where `ID` is the ID of the sensor that made the measurement, `(x,y)` are coordinates that specify the sensor location, `time` is the time of measurement,

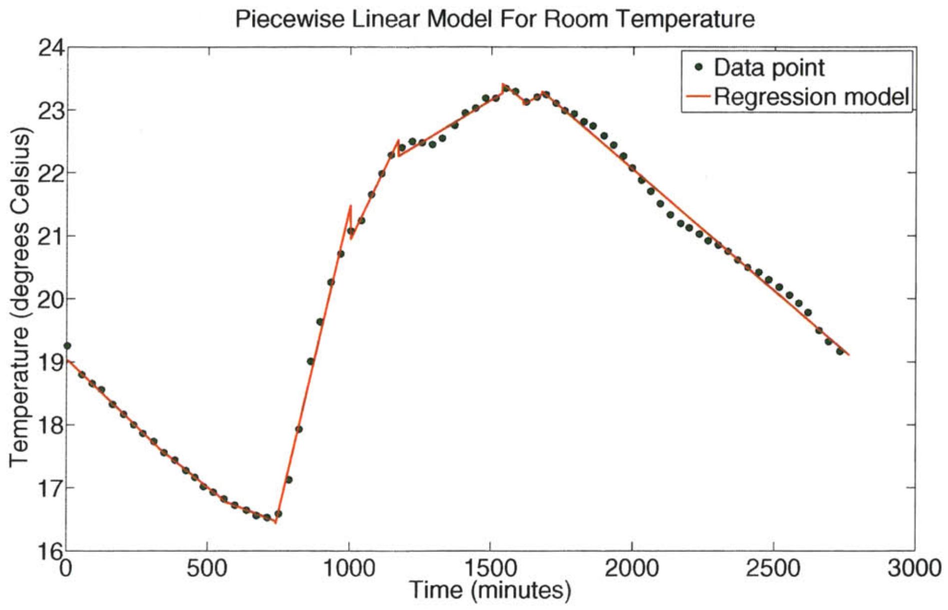


Figure 2-1: Temperature recorded by a sensor in the Intel Lab plotted as a function of time.

and `temp` is the actual temperature measurement.

The MauveDB system [7] introduces extensions to SQL to fit a regression view to data already in a relational table. The FunctionDB syntax is similar to that syntax. The FunctionDB query to fit a piecewise linear model of temperature as a function of time to the readings in `tempdata` looks like:

```
CREATE VIEW timemodel
AS FIT temp OVER time
USING FUNCTION Line2D
USING PARTITION FindPeaks, 0.1
TRAINING_DATA SELECT temp, time FROM tempdata
GROUP ON tid, x, y
```

This query instructs the DBMS to fit a regression model, `timemodel`, to the data in `tempdata`, using the column `temp` as the dependent variable, and data from `time` as the independent variable. This model can be queried like any relational table with the schema <`temp,time`>.

The `USING FUNCTION` clause in the above query specifies the type of regression function to use to fit the data – in this case, Line2D, which represents a line segment in 2D space.

The USING PARTITION clause tells the DBMS how to *segment* (partition) the data into regions within which to fit different regression functions. This example uses FindPeaks, a very simple in-built segmentation algorithm that finds peaks and valleys in temperature data and segments the data at these extrema. The parameter (0.1) to FindPeaks specifies how aggressive the algorithm should be about segmenting data into pieces. The choice of segmentation algorithm also determines how the regression view is updated when raw data is inserted or modified.

Finally, the TRAINING_DATA clause specifies the data used to train the regression model, which in this case consists of the readings from the table tempdata. The GROUP ON clause specifies that different models should be fit to data with different trajectory identifiers or different locations *i.e.*, different tid, x or y. This clause also automatically implies that tid, x and y will appear as non-model attributes in the resulting regression view, timemodel.

The segmentation algorithm, choice of model, and parameters used for fitting that appear in the USING FUNCTION and USING PARTITION clauses have important consequences for query accuracy and performance. A segmentation algorithm that is more aggressive about segmenting data into pieces (to fit the data better) results in a less compact representation, and hence worse query performance, while a less aggressive algorithm can mean better query performance, but a poorer fit. An extremely aggressive algorithm can also, somewhat paradoxically, be a bad idea because it can result in overfitting — the extreme example being a line segment joining every successive pair of points in the dataset. For this reason, the choice of model and parameters is quite complex, and can often depend considerably on application-specific criteria, like generalization performance, or corroboration by external or theoretical evidence.

Since the most appropriate choice for the basis functions used by a regression model often needs to be data or application dependent as explained above, a restricted set of built-in functions (e.g. lines or polynomials) is unlikely to prove sufficient. Therefore, this thesis proposes a query processor that enables users to create and add new classes of regression functions (Section 4.1).

In general, the choice of regression model has three aspects: the segmentation algorithm used for partitioning the data, the form of basis functions used to fit the data in each segment (*e.g.*, linear, polynomial, periodic) and the actual model parameters (function coefficients). While automatically choosing the form of the basis functions may not be desirable for the reasons discussed above, in the case where a regression model is mainly being employed to fill in gaps in the raw data (rather to smooth over measurement noise), it is possible to automate the segmentation algorithm and the choice of model parameters based on the criteria of query performance and accuracy. Chapter 5 of this thesis is devoted to this issue, and proposes and evaluates an in-built segmentation algorithm, `BSearch`, that performs well on our datasets and applications, and generalizes to any function of a single independent variable that can be fit using linear regression (even if the basis functions are non-linear). This algorithm allows the user to specify an error threshold to control the accuracy of the resulting regression fit and trade it off against query execution performance.

In addition to fitting regression models in a single pass over raw data as in the above view definition query, FunctionDB supports online model maintenance as new data is inserted into the system. Currently, our system supports online maintenance for one-dimensional time series data, for the case where all inserts are appends to the time series. Again, this feature is discussed in more detail in Chapter 5.

2.1.3 Queries

We now turn to querying regression models. We assume that a regression view `timemodel` for temperature as a function of time has already been constructed using `CREATE VIEW` as described previously. Given these views, we show below what each of the queries posed earlier look like in FunctionDB:

Query 1: Temperature at a given time (*Simple Selection*)

```
SELECT temp FROM timemodel WHERE ID = given_id AND time = 125
```

Query 2: Temperature above a threshold (*Simple Selection*)

```
SELECT time, temp FROM timemodel
```

```
WHERE ID = given_id AND temp < 18 GRID 0.5
```

Query 3: Time window average of temperature (*Aggregation*):

```
SELECT AVG(temp) FROM timemodel
WHERE ID = given_id AND time ≥ t1 AND time ≤ t2
```

Query 4: Distribution of temperatures experienced by a particular location (*Grouping, Aggregation*):

```
SELECT temp, AMOUNT(time) FROM timemodel
WHERE ID = given_id AND time ≥ t1 AND time ≤ t2
GROUP BY temp GROUPSIZE 1
```

All of the queries listed above use the regression fit to predict the values of the dependent variable used in the query. For example, Query 1 would return a temperature value for the time instant 125 even if the sensor reading for this time instant were missing, by evaluating the regression function at that time instant.

Also, some queries include a GRID clause. This is a simple SQL extension that specifies the granularity with which results are displayed to the end user. Since FunctionDB represents regression functions symbolically and executes queries without actually materializing the models at any point in a query plan (Chapter 4), the results of queries are *continuous* intervals, unlike a traditional RDBMS where queries return *discrete* tuples. Hence, most FunctionDB queries require a special output operator to display results to the user. GRID is one such output operator. For a query that returns a continuous result, in order to be compatible with a traditional discrete DBMS, GRID discretizes the independent variable at fixed intervals to generate output tuples in the vein of a traditional DBMS. For example, the `GRID 0.5` clause in Query 2 specifies that times where the temperature is below 18^o should be output as a sequence of tuples separated by a spacing of 0.5 along the `time` attribute.

The histogram query (Query 4) includes a GROUPSIZE clause which in this context indicates the bin size of the histogram. In other words, `temp` is grouped into bins of size 1^oC and the aggregate `AMOUNT(time)` is computed over these bins.

2.2 Spatial Queries on Car Trajectories

2.2.1 Application Scenario

Our second application aims to support queries on data from Cartel [8], a mobile sensor platform developed at MIT that has been deployed on automobiles in and around the Boston area. Each car is equipped with an embedded computer connected to several sensors as well as a GPS device for determining vehicle location in real time. The Cartel node can also be attached to sensors that collect different kinds of geographic data. The system has been used for over a year to collect, visualize and analyze diverse kinds of data including vehicle trajectories, WiFi connectivity, and road surface conditions.

In this thesis, we consider queries on car trajectory data specified by a sequence of GPS readings. This data requires interpolation because GPS data are sometimes missing, when the car passes under a bridge or through a tunnel, or whenever the GPS signal is weak. Also, while GPS values are usually quite accurate, they can sometimes include outliers, requiring filtration. This can happen due to multipath reflection *i.e.*, multiple copies of the GPS signal reflected from buildings or obstacles that interfere at the receiver.

The fact that trajectories correspond to cars driving on a structured road network, as opposed to being arbitrary curves, makes this data an attractive target for modeling. Using FunctionDB, we have constructed a piecewise regression model where the pieces are linear functions that represent road segments. This approximates the trajectory quite well, and smooths over gaps and errors in the data. Figure 2-2 illustrates real GPS data collected from a car driving near Boston, and the corresponding regression model. Notice the large gap in data near the Massachusetts Turnpike, and the linear function that helps interpolate this gap.

2.2.2 FunctionDB View Definition and Queries

The FunctionDB schema for the regression 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. Below, we

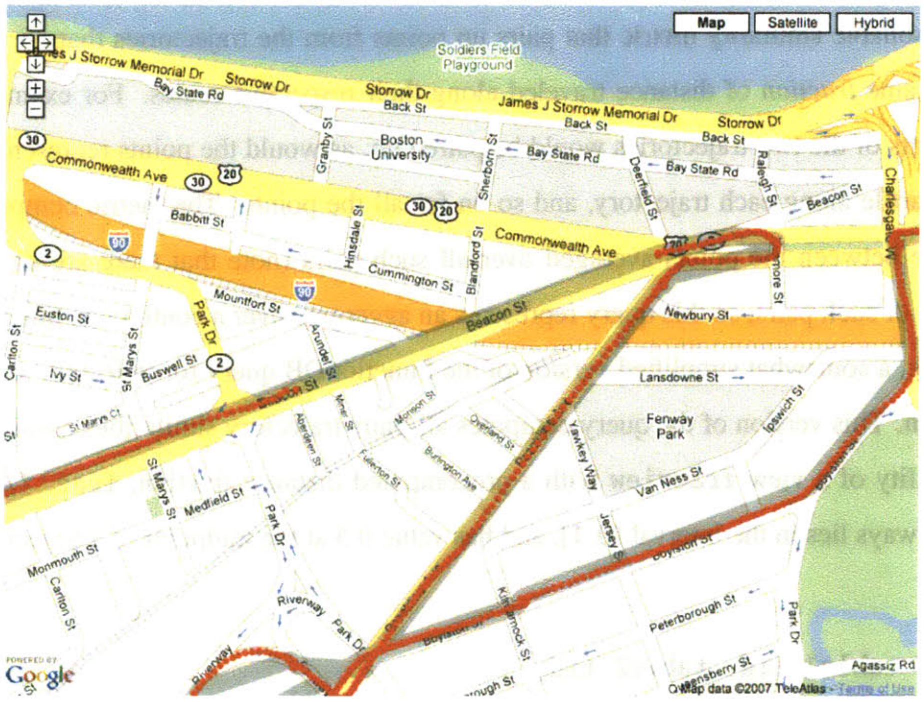


Figure 2-2: Trajectory of a car driving near Boston. The points represent raw GPS data, and the line segments illustrate our regression fit to interpolate the data.

describe two queries that Cartel users are interested in, and show how they are written in FunctionDB.

Bounding Box. How many (or what fraction) of the trajectories pass through a given geographic area (*e.g.*, bounding box)? This query is easy to express as a selection query on the view, say `locmodel`:

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

Trajectory Similarity. One application of the Cartel data is 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. For illustration, we use a simple, but reasonable similarity metric that pairs up points from the trajectories that correspond to the same *fraction* of distance traveled along their respective routes. For example, the midpoints of the two trajectories would be paired up, as would the points representing the first quartile along each trajectory, and so on for all the points. The metric computes the distance between the points averaged over all such pairs (note that there are an infinite number of such pairs, so this query represents an aggregate over a continuous function).

Here, a somewhat simplified version of the FunctionDB query for trajectory similarity is shown. This version of the query computes all-pairs trajectory similarities assuming the availability of a view `fracview` with a precomputed distance fraction, `frac`. Note that `frac` always lies in the interval [0, 1], and has value 0.5 at the midpoint of a trajectory.

```
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
```

As in the sensor application, querying the regression model, as opposed to the raw data, results in significant benefits. For example, even if the bounding box in Query 1 happened to fall entirely in a gap as shown in Figure 2-3, the trajectory would count as passing through the box as long as one of the underlying line segment(s) in the regression fit intersected it.

As the above examples illustrate, FunctionDB provides a *logical* abstraction to the user that is similar to a traditional relation in a DBMS. This has the advantage that SQL queries already written for raw data run in FunctionDB with little or no modification. However, because the underlying implementation represents and queries regression functions, the same queries can now tolerate missing or incorrect data. Also, as we show in this thesis, because

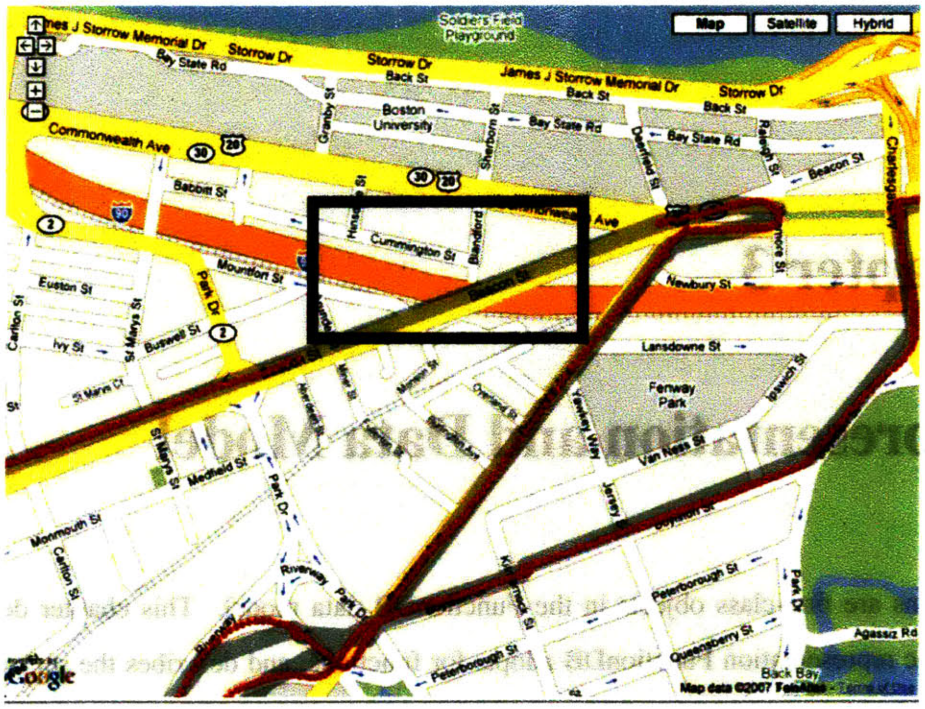


Figure 2-3: Bounding box query on trajectory data benefits from regression.

FunctionDB represents and queries models algebraically, as opposed to materializing the models into gridded data, the same queries also execute faster and more accurately than in a system that maintains an interpolated (gridded) representation of the regression model.

Also, while the idea of query processing on functions is particularly useful for supporting regression models, the idea itself is quite general. For example, an underlying model of trajectories based on actual road segment data (*e.g.*, from geographic data) could be used in FunctionDB and would work equally well, or better, than a model based purely on regression. The algorithms we present for query processing in FunctionDB can be used to represent and query this data, even though not it does not come from a "regression model".

Chapter 3

Representation and Data Model

Functions are first-class objects in the FunctionDB data model. This chapter describes the basic representation FunctionDB adopts for functions, and describes the semantics of relational and aggregate operations supported over functions in the system. In this thesis, we focus on functions of a single variable, but later discuss how algebraic query processing might extend to functions of more than one variable (Section 3.4).

Also, while FunctionDB can execute a wide and useful class of relational queries on functions of a single variable, we shall see that it is impossible to guarantee that symbolic query evaluation can be used for arbitrary functions derived from query expressions. Accordingly, in the general case, queries in FunctionDB may require approximation. The semantics of this approximation are defined and spelled out in this chapter.

3.1 Piecewise Functions

In the standard relational data model, data is stored in the form of discrete records, or *tuples*, each with multiple attributes (*fields*) in tables (*relations*). The data model defines several operators, termed *relational operators*, that operate on one or more of these tables, including σ (select), π (project) and $\bowtie$ (join). The relational model has become the data model of choice for most modern DBMSs, mainly because it was the first data model that enabled satisfactory physical data independence *i.e.*, separation of the logical semantics of queries from the physical mechanism used to implement them. A detailed overview of the relational model is found in [9].

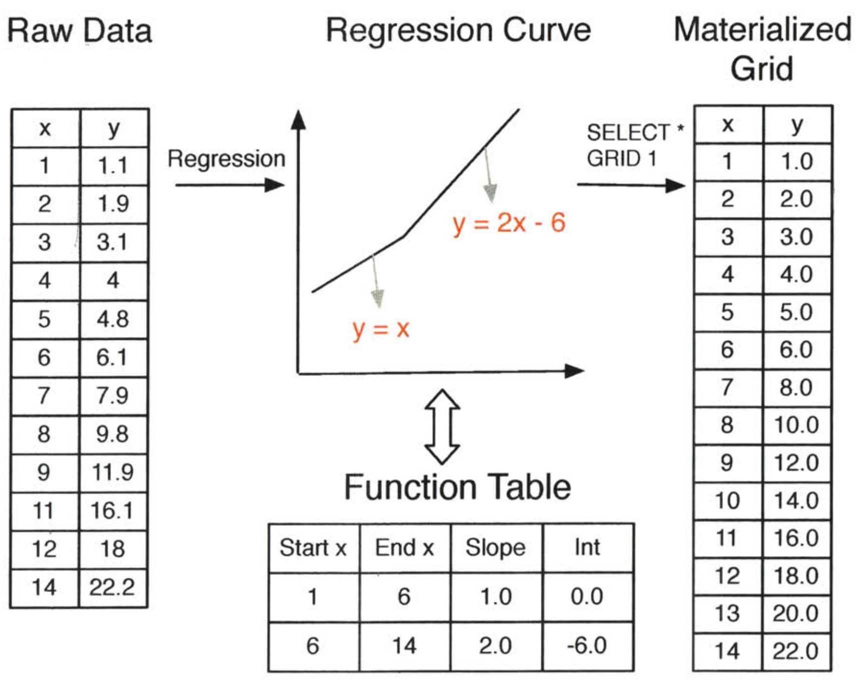
The basic idea behind the FunctionDB data model is to add a new type of queryable relation to the standard relational data model, which we term the *function table*. A function table represents regression models that consist of a collection of *pieces*, where each piece is a continuous function defined over an interval of values taken by the independent variable of regression.

Each tuple in a function table describes a single piece of the regression model, and consists of two sets of attributes: *interval attributes* that describe the interval over which the piece is defined, and *function attributes* that describe the parameters of the regression function that represents the value of the dependent variable in that interval. These parameters are in algebraic form; for example, in polynomial regression they are a list of polynomial coefficients.

In this and the discussion that follows, we will focus on function tables that only contain model variables, dependent or independent. However, in practice it is a must to support relations that also include other variables not part of a regression model. For example, the trajectory id `tid` in the trajectory example described in Section 2.2, Chapter 2. Note that non-model variables stored as part of a function table are restricted, in the sense that they must appear in the `GROUP ON` clause when fitting (Chapter 2): this ensures that each model piece has exactly one value for these attributes. Other attributes in the raw data which are not being interpolated, and do not appear in the `GROUP ON` clause when fitting the model are stored with the original discrete table containing the raw data. For example, this would happen if we had a `speed` attribute for each raw data point in the trajectory schema, but did not want to fit a model to the values in this column.

Figure 3-1 shows an example of FunctionDB fitting a regression model to raw data with two attributes: x and y. The data has been modeled with two regression functions that express y (the dependent variable) as a function of x (the independent variable): $y = x$ in the case when $1 \leq x < 6$, and $y = 2x - 6$ when $6 \leq x < 14$. In the function table, attributes "Start x" and "End x" are interval attributes while "Slope" and "Intercept" are function attributes.

While a function table is physically a finite collection of functions, at the query lan-



Raw Data

x	y
1	1.1
2	1.9
3	3.1
4	4
5	4.8
6	6.1
7	7.9
8	9.8
9	11.9
11	16.1
12	18
14	22.2

Function Table

Start x	End x	Slope	Int
1	6	1.0	0.0
6	14	2.0	-6.0

Materialized Grid

x	y
1	1.0
2	2.0
3	3.0
4	4.0
5	5.0
6	6.0
7	8.0
8	10.0
9	12.0
10	14.0
11	16.0
12	18.0
13	20.0
14	22.0

Figure 3-1: Raw data, corresponding function table, and result of a `SELECT *` query.

guage level, a function table is a different kind of relation — one which describes an *infinite* set, and logically consists of every single point on a continuous curve representing the function. This is in contrast to traditional relations that are finite collections of discrete tuples. To illustrate this, Figure 3-1 also shows the result of a `SELECT *` query on the function table. Since it is not possible to output an infinite set of tuples, the query includes a `GRID 1` clause, which tells the system to *materialize* the function table by evaluating the function at intervals of 1 along the independent variable, x, and display the resulting (gridded) set of discrete points. As we shall see, FunctionDB can compute the answers to many relational queries symbolically, and hence many query plans only perform gridding at the very end for displaying the result to the user.

Interval attributes represent an interval on the real line in the case of a single independent variable. Our data model permits relations that contain overlapping intervals or regions of independent variable(s). This enables us to support models in which there can be multiple values of the dependent variable corresponding to a single value or set of values taken by the independent variable(s). While regression functions on time series data are single-valued, this relaxation is useful to represent other models that are not single-valued, like car trajectories.

In our implementation, we group function attributes into a function ADT that implements algebraic primitives like equation solving, inversion and definite integrals. Operators in our algebraic query processor are implemented entirely in terms of these primitives (Table 4.1 in Chapter 4). This has the advantage that our operator implementations are extensible to a wide variety of functions, as well as to user-defined regression models.

3.2 Relational Algebra and Query Semantics

Since function models are just relations, traditional relational operators such as selections, projections, joins, as well as aggregate operators (like averages and counts) extend naturally to function tables. These operators retain semantics that are similar to their discrete versions, but with some differences, because our operators logically operate over *infinite* relations, as opposed to traditional operators on finite relations. For example, aggregate operators need to be generalized from discrete sums/counts to definite integrals in the continuous domain. In this section, we first discuss the semantics of query results in FunctionDB, and then go on to define the semantics of individual operators on infinite relations.

3.2.1 Semantics of Query Results

There do exist relational queries which cannot be solved entirely in the functional domain in our framework. For instance, computing an aggregate of an arbitrary expression involving the columns of a function table (*e.g.*, the mean square difference of two functions) may not be possible to do symbolically if FunctionDB does not know to integrate the derived function analytically; in this case, the system must fall back to approximating the query result using numeric integration.

For this reason, FunctionDB adopts a *graded* semantics for query results. At one extreme is a pure gridding approach (like that adopted in the MauveDB work). In this approach, the first step in a query plan always consists of applying a GRID operator to discretize the function tables being operated on by the query. The semantics of the GRID operator are as follows: GRID takes a function table as input and selects a finite set of tuples, sampled at a discrete interval from the corresponding infinite relation by evaluating the function at fixed intervals of each independent variable. For example, the result of applying GRID with size 1 to a table with two pieces, $y = 2x$ for $0 \leq x < 2$ and $y = x + 2$ for $2 \leq x \leq 4$, is a finite collection of tuples: $\{ (0,0), (1,2), (2,4), (3,5), (4,6) \}$. The grid spacing for discretization can be controlled by the user with the GRID clause, an extension to SQL supported by FunctionDB. A narrower spacing implies more accurate query execution, and conversely a wider spacing means less accuracy.

A pure gridding approach has the advantage that it is always guaranteed to be usable, and can yield reasonable answers to queries. In practice, however, most queries lend themselves at least partially to symbolic evaluation. The process of converting a pure gridding query plan to use algebraic processing can be logically thought of as a "lift-up" process, where the strategy is to lift the GRID operators progressively up the query plan until the system encounters an operator that cannot be converted to use functional processing. In the best case, the query can be executed completely in the functional domain, and the only GRID operator that remains is at the top of the query plan, for displaying results. The lift-up process is always guaranteed to improve the accuracy of query execution.

As we show in Chapter 4, some important classes of relational queries can be executed entirely in the functional domain on functions of a single variable:

- Filters on individual attributes of the relation (*e.g.*, of the form $y > 3$ or $x > 2$).

- Filters on algebraic expressions (*e.g.*, $x + y > 2$) where the type of the expression can be inferred at query compile time, and is known to the FunctionDB system. For the special case of polynomial functions, for example, FunctionDB can handle arbitrary expressions involving the $+$, $-$ or $\times$ operators entirely in the functional domain.

- Equijoins between function tables, on either the independent or dependent attribute(s) of either table (*e.g.*, of the form $R1.x = R2.y$), provided the join condition can be rewritten as a condition on a function of a single variable (of the form $F(x) = 0$) whose type is known to FunctionDB.

- Aggregate queries on algebraic expressions whose type is known, and for which an indefinite integral is known to FunctionDB (some functions, like $\frac{\sin x}{x}$ cannot be integrated analytically in closed form).

Also, parts of more complex queries which cannot be executed without approximation can still take advantage of algebraic query execution if they fall into one of the above categories, each of which is a commonly occurring building block in many relational queries.

3.2.2 Semantics of Individual Operators

3.2.3 Selection

The selection operator, σ_P, when applied to the infinite relation F represented by a function table yields another relation F', that can also be represented by a function table. F' is defined to be the (largest) subset of F satisfying the selection predicate P. Selection predicates can involve both dependent and independent variables of regression.

As an example of selection, applying $\sigma_{y \geq 5}$ to a table with two pieces: $y = 2x$ for $0 \leq x \leq 2$ and $y = x + 2$ for $2 < x \leq 4$ results in a function table with a single piece: $y = x + 2$ for $3 \leq x \leq 4$. The first piece, $y = 2x$ for $0 \leq x \leq 2$, was discarded because no y value predicted by the model in this region can exceed 5. Similarly, only a part of the second piece was retained by the σ operator.

The result, F' of selection can be either a finite or infinite set depending on the predicate. For example, an equality (=) predicate on the dependent variable of a linear function will always produce a finite result (except in the special case of a constant function) while an inequality predicate, like $\geq$ in the example above, will produce either no result, or a result that represents an infinite set.

In cases where the function table has overlapping intervals over which functions are defined (this happens with multi-valued relations, like car trajectories) the result of σ_P is the union of the results obtained by applying σ_P individually to each interval in F.

3.2.4 Projection

The projection operator, π_V when applied to an infinite relation F yields an infinite relation F' consisting of all the possible values the projection attributes V take in the original relation F. If the variables being projected are all independent or dependent variables, the result is simply a set of intervals. As in the case of selections, applying π to a function table with overlapping intervals yields the union of results from applying π to individual intervals in the function table.

3.2.5 Join

The join operator, $\bowtie_P$ is applied to two relations R_1 and R_2, where R_1 and R_2 can be function tables or normal relations. Joins have identical logical semantics to traditional relational algebra: a tuple (t_1, t_2) is present in the joined result if and only if $t_1 \in R_1$, $t_2 \in R_2$ and (t_1, t_2) satisfies the join predicate P. For example, consider a table F_1 containing the piece $y = 2x - 6$, defined over $x \in [4, 6]$, a table F_2 containing $y = x$ defined over $[3, 7]$, and the equijoin operator $\bowtie_{F_1.y=F_2.y}$ applied to the two tables. The result of the join is finitely representable as a function table containing two dependent variables, one for each of the x attributes from the two tables, and a single independent variable which is the equijoin attribute y. The table contains a single piece given by $x_1 = \frac{1}{2}y + 3, x_2 = y$, defined over $y \in [3, 6]$.

Note that it is possible to join a function table to a traditional relational table. In the special case of equijoins, the result will typically be a finite set of tuples.

Also, the results of some joins (*e.g.*, cross products of completely unrelated functions, with no join predicate) are not function tables, necessitating the use of gridding as a fall back option to execute the query.

3.2.6 Aggregate Operators

In addition to relational operators, our algebra includes a suite of aggregate operators that are the continuous analogues of traditional database aggregates like `SUM`, `AVG` and `COUNT`.

SQL Aggregate	FunctionDB Analogue
COUNT($R.A$) = $\Sigma_{v \in R.A} 1$	AMOUNT($R.A$) = $\int_{v \in R.A} 1\, dv$
SUM($R.A$) = $\Sigma_{v \in R.A} v$	AREA($R.A$) = $\int_{v \in R.A} v\, d(R.X)$
AVG($R.A$) = $\frac{\Sigma_{v \in R.A} v}{\Sigma_{v \in R.X} 1}$	AVG($R.A$) = $\frac{\int_{v \in R.A} v\, d(R.X)}{\int 1\, d(R.X)}$

Table 3.1: Discrete aggregates and their FunctionDB counterparts.

Traditional aggregates over discrete relations generalize to definite integrals when operating over continuous functions. For example, consider the SQL COUNT operator applied to an attribute $R.A$ of a relation R. This operator counts the number of tuples that occur in column A of R, which essentially computes the sum $\Sigma_{v \in R.A} 1$ (ignoring duplicates). In the continuous domain, this sum generalizes to a definite integral: $\int_{v \in R.A} 1\, dv$. If $R.A$ represents time, then this integral computes the total time spanned by the model R, which is the sum of lengths of all the time intervals that pieces of R are defined over. We name this aggregate operator AMOUNT.

Other aggregate operators generalize similarly. Table 3.1 shows three common SQL aggregate operators, the discrete sum they compute and the definite integral that this sum generalizes to in the continuous domain. In the table, $R.X$ denotes the independent variable(s) that the integral is computed over. The aggregation attribute $R.A$ can be either a dependent or independent variable.

3.3 Discussion

Our choice of piecewise continuous functions as the underlying representation for models has several benefits:

Algebraic Query Execution. Restricting models to functions enables query execution using algebra (Chapter 4), which helps answer queries faster and more accurately.

Losslessness. It is a completely lossless representation of the underlying continuous function. Because the FunctionDB query processor usually avoids materialization until late in a query plan, there are no errors introduced in addition to the inherent error of fitting the model. In contrast, executing queries on a gridded representation, as done in MauveDB [7], introduces and propagates additional discretization errors throughout the query plan. (Chapter 4.4).

Reduced Footprint. It is compact compared to maintaining a gridded representation of the model, because a function table has only as many rows as there are pieces in the regression model, usually an order of magnitude smaller than a materialized representation. Lower footprint results in substantial savings in I/O and CPU cost, and order of magnitude better performance on queries (Chapter 4.4).

Wide Applicability. It is simple to reason about and implement, and at the same time generalizes naturally to support a wide class of regression models used in practice.

Function tables are only one possible way to represent an infinite relation using a finite set of tuples. For example, our data model is more restrictive than constraint databases [20, 21], which represent infinite regions like polygons or line segments as algebraic constraints. Constraint databases are more general than function tables because any function table can be described by a set of inequality constraints for interval attribute(s), like $740 \leq t \leq 1001$, and an equality constraint for functions like temp = $2t + 3$. These databases are capable of solving arbitrary linear programs, and the results of arbitrary relational operators in higher dimensions are always well defined, but this generality comes at a price. General constraint solvers are complex to build, with the result that in practice, constraint databases have been confined to linear constraints. Our query processor is simpler and more extensible to a variety of functions.

3.4 Future Work: Higher Dimensions

A simple function-only representation does not directly generalize to functions in higher dimensions — in other words, functions of more than a single variable. While functions of multiple variables can be represented in our system and stored in function tables, the results of relational operators on functions of more than one variable are not always function tables, even for simple operators like filters and projections. For example, the selection predicate $\sigma_{z<4}$ when applied to the function $z = x^2 + y^2$ yields the circular region $x^2 + y^2 < 4$, which is not representable as a function table.

We hypothesize that combining our representation and framework for functions (Section 4.1) with the expressive generality of the constraint data model (see above) might help tackle this problem. Interval attributes would generalize to describing a region, and constraints would be expressed as conditions on functions (*e.g.*, $f(x,y) < 4$, where $f(x,y) = x^2 + y^2$, in the above example). The system would still require approximation to actually evaluate queries involving complex regions defined by constraint boundaries: however, since any higher dimensional region can be approximated with a set of hypercubes, this approach still has the promise of being more efficient than simple gridding. We leave the implementation of higher dimensional functions in FunctionDB to future work.

Chapter 4

Query Processor

The core of FunctionDB is an algebraic query processor that executes relational queries using operations in the functional domain. This chapter describes the key idea behind our query processor: an extensible framework for expressing relational operators as combinations of primitive operations on functions, and spells out the algorithms for relational operations in FunctionDB in detail. This is followed by a detailed experimental evaluation of our query processing algorithms, and comparison to the existing gridding-based approach. The evaluation uses real data and queries from the applications introduced and described in Chapter 2.

Note that this chapter focuses on algorithms for query processing on regression functions that have already been fit to data; the next chapter describes and evaluates algorithm(s) for online model maintenance as new raw data is inserted into FunctionDB.

4.1 Extensible Function Framework

The obvious way to implement a function-aware query processor would be to define abstract data types (ADTs) for different classes of functions (*e.g.*, linear, periodic or polynomial) and implement relational operators that work on each function type. While this approach would be adequate if FunctionDB only needed to support a limited class of functions, it requires replicating implementations of operators for each new class of function added to the system.

Hence, we have instead chosen to characterize a small set of abstract "interfaces" implemented by *all* continuous functions of a single variable, and express relational and aggregate operators in terms of these interfaces. Since our operators work by invoking these interfaces and do not directly rely on the type of function, it is sufficient to implement these algebraic primitives for each new model type that needs to be supported.

As discussed in Chapter 3, at one extreme, it is possible to implement arbitrary relational queries on functions using a pure gridding approach. The only property required for this approach to work is *function evaluation, i.e.*, the ability to compute the value taken by the function, $f(x)$ for a particular value x of the independent variable. However, this approach is quite slow and results in loss of accuracy due to discretization (Section 4.4).

Suppose we now require functions to implement a simple new primitive: *root finding*. Given a real number a as input, a "root finding" interface determines all values of the independent variable x for which $f(x) = a$. This primitive is sufficient to solve selection queries with predicates of the form[1] `WHERE` $y = a$, which involve the dependent variable of regression, y.

Selection based on root finding is simple. For each piece in the function table, the algorithm invokes root finding to determine candidate values of x such that $f(x) = a$, and then checks which of these values fall within the interval of x over which the piece is defined. For each such value, say x_0, it outputs a tuple with the same function coefficients as the original tuple, but defined over a *single point*: $[x_0, x_0]$. If none of the candidate x values lie within the interval of definition, this means that no points in the input piece satisfy the selection predicate, so no output tuples are emitted.

Selection by root finding is the simplest example of algebraic query processing. If more properties are available for a class of function, taking advantage of them *progressively* enables new classes of operators to be executed algebraically without having to fall back on gridding. To illustrate, Table 4.1 lists some common classes of relational operators, and algebraic primitives that would enable executing that operator without materialization. The table is not exhaustive, but meant to give a flavour of the algebraic techniques used by

[1]Because pieces are continuous functions, it actually turns out that root finding is sufficient to solve arbitrary $>$ / $<$ predicates (Algorithm 1).

Relational Operator	Application Example	Required Primitives	Advantages Over Gridding
Selection on dependent variable $(\sigma_{F.y>a/F.y<a/F.y=a})$	Find when temperature crosses threshold	Root Finding, Evaluation	Faster if low selectivity
Selection on independent variable $(\sigma_{F.x>a/F.x<a/F.x=a})$	Restrict query to time window	None (Interval Manipulation)	Faster if low selectivity
Equijoin of independent variables, compare dependent $(\bowtie_{F_1.x=F_2.x \vee F_1.y>/<F_2.y})$	Compare temperatures at two sensors at same time	Function Subtraction, Root Finding	More Accurate for = test, Faster for > / < test if low selectivity
Equijoin of dependent variables $(\bowtie_{F_1.y=F_2.y})$	Line up vehicle trajectories on distance fraction (Section 2.2, Query 2)	Function inversion	More accurate for = test, Much faster
Group by independent variable, aggregate dependent	Compute temperature average over time windows	Definite integral (Area under curve)	More accurate, faster
Group by dependent variable, aggregate independent	Duration histogram for temperature ranges (Section 2.1, Query 5)	Function inversion, definite integral for inverse	More accurate, faster
Selection on non-model variable	Restrict query to particular sensor(s) or trajectory(s)	None (traditional relational operator)	None
Aggregate non-model variable	Count trajectories within bounding box	None (traditional relational operator)	None

Table 4.1: Commonly used relational operators and algebraic primitives that can be used to execute them. F, F_1, F_2 denote function tables, x, x_1, x_2 denote independent variables, y denotes a dependent variable, and a, a_1 and a_2 denote constants (query parameters).

FunctionDB. The next section presents more detailed algorithms for each of the relational operators.

4.2 Expressions and Type Inference

In general, a relational query can involve arithmetic or algebraic expressions applied to the independent or dependent variables in a function table. For example, the selection query "SELECT x, y FROM F WHERE $x + y > 3$ involves a selection predicate applied to the algebraic expression $x + y$. In order to take advantage of algebraic processing to execute this query without gridding, FunctionDB needs to be able to determine the type of function represented by the algebraic expression $x + y$.

Our prototype implementation of FunctionDB uses the property-based framework defined in the previous section for rudimentary type inference: our system can be augmented extended with rules for different arithmetic and algebraic operators specifying their result types in terms of the types of their arguments. For example, it would be possible to add a rule that applying $+$, $-$ or $*$ with two polynomial functions as arguments always yields a polynomial function. These rules can be used for type inference at compile time.

FunctionDB does have to fall back on using GRID if it cannot infer a type at compile time, or if a property (*e.g.*, analytical integration) is not available for a derived/inferred function type. The more general problem of a consistent and comprehensive type system for algebraic expressions is beyond the scope of this thesis. However, computer algebra systems like Maple and Mathematica are capable of fairly sophisticated computations on a wide variety of functions: it should be possible to use a more sophisticated CAS framework in FunctionDB for type inference instead.

4.3 Operators

This section presents algebraic algorithms for the standard relational operators: selection, projection, join, grouping as well as declarative aggregate operators, as applied to continuous functions of a single variable.

Our operators all export and use an iterator interface typically used in implementations of relational DBMS systems. A query plan consists of a directed acyclic graph of operators, with the leaves being iterators over relational tables or function tables. Each operator exposes an interface that returns the next output tuple if available, or a null value if no more result tuples exist. As a simple example, a selection (filter) operator would return the next tuple from its child operator which satisfies the filter predicate on being queried by its parent operator in the query plan, or a null value if no more tuples exist satisfying the predicate. The top-level operator exposes the same iterator interface: a user program connecting to the DBMS would pull tuples from this interface to iterate over a query result.

For each operator, we also indicate the conditions under which algebraic processing is possible for that operator. These preconditions are checked at query compile time, rather than at execution time. If any precondition(s) for executing an algebraic version of an operator fail, the system falls back on gridding. This simply involves inserting a `GRID` operator before the operator's inputs, and substituting the operator with a traditional DBMS operator for the same task.

4.3.1 Selection

Selection queries can involve either the dependent or independent variable of a model, or both. Selection over the independent variable ($x > / < / = a$) is easy, and does not depend on the function at all. It merely requires checking each piece in the function table to determine what portion of the corresponding interval (if any) satisfies the selection predicate. For example, applying the predicate $x \geq 3$ to the function piece $y = 2x$ defined over $[2, 4]$ would yield as output a piece with the same function, $y = 2x$, but now defined over $[3, 4]$.

The previous section described selection for predicates of the form $y = a$ using root finding. Algorithm 1 handles predicates of the form $y > a$. The algorithm for $y < a$ follows by symmetry. This algorithm also uses root finding, and additionally exploits the fact that a continuous function $f(x)$ has alternating signs in the intervals of x between the roots of the equation $f(x) = 0$, provided "corner case" (degenerate) roots where the function is

Algorithm 1: (Selection on y) $\sigma_{y>a}(F)$

```
Note: Algebraic primitives are typeset in bold italics.
Given: An upstream iterator, $F$, to apply selection to, and a selection predicate,
        $y > a$.
Precondition: $F.y$ supports the findroots primitive.
if cache not empty then return tuple from cache
while $F$ has more tuples do
    Next ← $F$.GetNext()
    {Start, End} ← Next.interval
    Piece ← Next.function
    AllRoots ← Piece.findroots($a$)
    Candidates ← {x: x ∈ AllRoots and Start ≤ x ≤ End}
    SR ← Smallest root ∈ Candidates larger than Start
    if Piece.evaluate(Start) > $a$ then
        I ← Alternating intervals from Candidates including [Start, SR]
    else
        I ← Alternating intervals from Candidates excluding [Start, SR]
    for $i \in I$ do
        Output.function ← Next.function
        Output.interval ← i
        return Output if first tuple, else cache it
```

tangential to the line $y = a$ are counted twice. Also note that for general functions, a single piece could contain multiple disjoint intervals of x where the selection predicate is true (though there is at most one for linear functions).

The algebraic algorithm for selection presented above outperforms the gridding approach, because its running time depends only on the number of pieces in the regression model, which is usually an order of magnitude smaller than a materialized grid of points. Also, as our evaluation in Section 4.4 shows, even if the selection result ultimately needs to be materialized to display results to the user, the algebraic approach still has significant performance gains when the filter predicate is selective.

4.3.2 Join

This section describes algorithms for different classes of joins, classified on the basis of the type of join predicate:

- Equijoins on the independent variable between two function tables, corresponding to predicates of the form $F_1.x_1 = F_2.x_2$.
- Equijoins on the dependent variable between two function tables, corresponding to $F_1.y_1 = F_2.y_2$.
- Joins on predicates with algebraic/arithmetic expressions that can depend on independent or dependent variables from either function table.
- Joins with multiple predicates from one of the above categories.
- Joins whose results are regions that cannot be expressed as function tables (*e.g.*, cross products).
- Joins between function tables and standard discrete relational tables.

While we focus on the first four categories of joins (they appear commonly in applications of interest, and in the examples discussed in this thesis), we later overview how joins in the last two categories are implemented.

Equijoin on x

As with selections, equijoins between two function tables involving only the independent variable are easy to compute and do not use the function attributes at all. Any of the traditional join algorithms used in a relational DBMS can be used here, with a minor modification to test for overlap of x intervals rather than to compare attributes. For example, a simple nested loops join would loop over all pairs of pieces in both relations and determine which pairs overlap. For each pair of pieces whose x intervals overlap, the algorithm outputs a result tuple whose interval is the intersection of the overlapping x intervals, and which contains the function attributes from both pieces, copied over without any modifications.

Equijoin on y

Equijoins between two function tables on the dependent variable are a little trickier. The key insight is that an equijoin on y can be transformed to an equijoin on x using function inversion. As a simple example, consider the problem of joining two isolated pieces: $y = 2x + 2$ defined over $x \in [0, 2]$, and $y = x$ defined over $x \in [3, 7]$. Inverting the linear functions enables us to rewrite x as a function of y instead. The transformed equivalents of these pieces are $x = \frac{1}{2}y - 1$ defined over $y \in [0, 6]$, and $x = y$ defined over $y \in [3, 7]$. We now use the procedure for equijoins on the independent variable, yielding a composite piece with two functions: $x = \frac{1}{2}y - 1$ and $x = y$, *both* defined over the overlapping range $y \in [3, 6]$. It is easy to verify that this correctly represents the result of the y equijoin.

While the above procedure works for functions that have a unique inverse (like linear functions) it may not be immediately obvious how to extend the algorithm to functions that may not have a mathematical inverse *i.e.*, functions like $y = x^2$ for which there are multiple values of x that yield the same value of y. Here, our piecewise data model allows us to provide well defined answers to such queries. For such functions, we require the inversion primitive to return a *list* of inverse pieces defined over appropriate intervals, rather than a single piece. For example, invoking inversion on the quadratic function $y = x^2$ defined over $x \in [-2, 2]$ would return *two* pieces: $x = +\sqrt{y}$ and $x = -\sqrt{y}$, both defined over the interval $y \in [0, 4]$. For periodic functions defined over a specific interval (*e.g.*, trigonometric functions), the inverse primitive might need to return multiple pieces depending on the length of the interval. Algorithm 2 formally describes this procedure.

Joins On Algebraic Expressions

Join predicates can involve complex algebraic expressions that depend on both the independent and dependent variables of a model. For example, one of the join predicates in Query 3 (Chapter 2, Section 2.1): `ABS(T1.temp - T2.temp)` > 5, involves a subtraction followed by applying an absolute value operation. In this case, the query is decomposed into two stages: a preliminary *map* operator that computes the function representing `T1.temp - T2.temp` (this being possible thanks to the equijoin predicate on `time`), and a selection operator applied to the resulting expression.

As mentioned in Section 4.2, evaluating arbitrary complex algebraic expressions over functions is a hard problem. For one, an algebraic expression can result in a derived function type that may not be known to the system, or may not support required primitives for

Algorithm 2: (NL-Join on y) $\bowtie_{F_1.y_1=F_2.y_2}$ (F_1, F_2)

Given: Upstream iterators F_1 and F_2, and join predicate $F_1.y_1 = F_2.y_2$.
Precondition: $F_1.y_1$ and $F_2.y_2$ support the *invert* primitive.

```
if cache not empty then return tuple from cache
while $F_1$ has more tuples do
    NextOuter ← $F_1$.GetNext()
    OuterInv ← NextOuter.function.invert()
    OuterInt ← OuterInv.interval
    $F_2$.Rewind() ; // Reset inner iterator
    while $F_2$ has more tuples do
        NextInner ← $F_2$.GetNext()
        IList ← InnerPiece.function.invert()
        for InnerInv ∈ IList do
            InnerInt ← InnerInv.interval
            if OuterInt and InnerInt overlap then
                Output.interval ← OuterInt ∩ InnerInt
                Output.function1 ← OuterInv.function
                Output.function2 ← InnerInv.function
                return Output if first tuple, else cache it
```

query processing. We do not claim that FunctionDB provides a complete end-to-end solution to this problem. Rather, our aim is to provide an extensible framework to add more types and properties to the system so that queries can benefit incrementally as more types are supported.

Joins Requiring Approximation

As discussed previously, some joins of completely unrelated function tables translate into higher dimensional regions, not expressible in our simple function data model. For example, cross products of unrelated function tables with no restricting predicate fall into this category, as do joins with only > and < predicates imposing constraints on the combined tuple. These joins also require gridding.

Complex Predicates

Many joins involve a predicate which is a logical (Boolean) expression formed from simpler predicates that fall into one of the above categories. For example, consider the join in

Query 3 mentioned earlier in the context of the temperature sensor application (Chapter 2, Section 2.1):

Query 3: Nearby locations reporting differing temperatures (*Join*)

```
SELECT T1.time, T1.ID, T1.temp, T2.ID, T2.temp
FROM timemodel AS T1, timemodel AS T2
WHERE T1.time = T2.time AND
ABS(T1.temp - T2.temp) > 5 AND
(T1.x - T2.x)^2 + (T1.y - T2.y)^2 < 2^2
```

The above query is a self-join between parts of the same function table corresponding to temperature readings from two different sensors. The join predicate here is the logical conjunction of three predicates: an equality predicate on the independent variable of regression (`T1.time = T2.time`), and two other predicates involving algebraic expressions over the independent and dependent variables. The problem of constructing appropriate query plans for such joins (*e.g.*, deciding which predicates to execute first) is similar to query optimization in existing RDBMSs, and we leave this as an interesting area for future work.

Joins With Normal Tables

Joins between traditional relations (containing discrete tuples) and function tables are essential to pose "mix and match" queries between the raw data (preserved in a discrete table) and a regression model that has been fit to the data. For example, a query to evaluate the fit error of a model in a specific window of time or range of the fit could be expressed as such a join query.

For functions of a single independent variable, joins with normal tables can be executed without approximation. For example, an equijoin query on a particular attribute (say x) can be executed by simply evaluating each of the pieces in the function table at each of the values of x in the normal table, and testing the join predicate for each candidate pair of records in the two tables. Similarly, other kinds of join predicates can be evaluated by *substitution*: values from the normal table are substituted into the join predicate to create a succession of selection predicates on the function table, whose results are concatenated together to yield the query result. For example, consider the join predicate $F_1.y_1 = T_2.y_2$. If the values of y_2 occurring in T_2 are $v_1, v_2, v_3, \ldots$ then for each such value, this would translate into evaluating selection predicates of the form $F_1.y_1 = v_1$, $F_1.y_1 = v_2, \ldots$

4.3.3 Indexing

While the discussion has so far focused on the simplest algorithms for relational queries (*e.g.*, NL joins), as in a traditional DBMS, FunctionDB can also use indexes to speed up queries.

Selections and joins on x need to look up intervals of x overlapping with a given point or interval, and therefore benefit from an interval tree index that stores intervals of x. For selection or join predicates involving the dependent variable y, FunctionDB can automatically build a similar index for continuous functions that can be differentiated symbolically. The idea is to determine the extreme values of a function $f(x)$ in its interval of definition by locating the points where its derivative, $f\prime(x)$ vanishes. The maximum and minimum values taken by y in each function piece are stored as intervals in a tree. Because a continuous function $f(x)$ defined over an interval I is guaranteed to take all the values between its extreme values in I, the interval tree index can be used for a range or equality query. The desired range/point is looked up in the index to find intervals overlapping it, and within each of those intervals, it is easy to use the *findroots* primitive on $f(x)$ to determine the result tuples for the join or selection.

As in traditional RDBMSs, if a selection/join query is selective and not many pairs overlap, it should be faster to look up overlapping intervals in the index.

4.3.4 Aggregate Operators

FunctionDB includes implementations for the three aggregates mentioned in Section 3: namely, `AMOUNT` (analogous to SQL `COUNT`), `AREA` (analogous to SQL `SUM`) and `AVG`. Just as with select-project-join queries, these aggregate functions can be used to compute aggregates of fields that are independent variables, dependent variables or algebraic expressions.

We name them slightly differently from traditional SQL aggregates because they operate in the continuous domain.

In the continuous domain, aggregates over any kind of variable (independent or dependent) can be evaluated as a definite integral, which is equivalent to the limit of a sum.

For example, consider the aggregate `AMOUNT(x)` applied to tuples from a function table F with schema <x, y=f(x)>. For each tuple fed to it with x interval $[x_1, x_2]$, `AMOUNT(x)` computes the length of the interval $x_2 - x_1$, and accumulates the sum of these lengths[2] over all the input tuples in F. Mathematically, `AMOUNT(x)` can be viewed as computing the sum $\Sigma_{[x_1,x_2]\in F}\ (\int_{x_1}^{x_2} 1\ dx)$ (though this expression ultimately simplifies to the length of the interval, as shown above).

While aggregates over the independent variable have simple expressions and do not actually require computing symbolic integrals, this ability is required for computing averages or sums of dependent variables. For example, the aggregate `AREA(y)` when applied to a regression model with schema <x, y> computes the total area under the regression curve, which is equal to the sum of these areas under all the pieces in the function table. For an individual piece $y = f(x)$ defined over the interval $[x_1, x_2]$, the area under the piece is given by the definite integral $\int_{x_1}^{x_2} f(x)\ dx$. Accordingly, our aggregate implementation works by invoking a "definite integral" primitive on each piece in the function table. This primitive takes the endpoints of the interval as a parameter, and returns the value of the integral (which itself is computed symbolically). `AREA(y)` accumulates the returned values over all the pieces in the table, in effect evaluating the sum $\Sigma_{[x_1,x_2]\in F}\ (\int_{x_1}^{x_2} y\ dx)$.

The other aggregates: `AREA(x)`, `AVG(x)`, `AMOUNT(y)`, and `AVG(y)` have similar expressions, derived from Table 3.1 in Section 3.

Numerical Integration. It is not always possible to use symbolic integration to execute aggregate queries: some functions cannot be integrated symbolically (like $\frac{\sin x}{x}$). The fallback option for executing aggregate operators, as with selections and joins, is to use gridding. The discretization (`GRID`) operator is first used to materialize the function(s) into discrete tuples, and the aggregate sum (or count) is computed as a discrete sum or count. Using

[2]It is possible to either include or exclude the signs of the individual integrals when computing this sum. We have implemented both versions of aggregate operators.

gridding for aggregation is in effect equivalent to using numerical integration to compute the value of an integral.

4.3.5 Grouping Operators

As with other operators, aggregate queries that first `GROUP BY` a model attribute can be classified on the basis of the grouping field type: independent or dependent.

Grouping on the independent variable into groups of a given size, S, is accomplished with simple interval manipulation. The algorithm first splits each of the input tuples into tuples defined over smaller intervals that entirely lie within an interval of the form $[kS, (k+1)S]$ for some integer $k \geq 0$. For example, if using bins of size 0.1, the intervals $[0.22, 0.38]$ and $[0.38, 0.56]$ would be split into $[0.22, 0.3]$, $[0.3, 0.38]$, $[0.38, 0.4]$, $[0.4, 0.5]$, $[0.5, 0.56]$. Once the tuples have been split, it is easy to hash each of the smaller tuples into an appropriate bucket based on the start points of their intervals (*e.g.*, $[0.3, 0.38]$ and $[0.38, 0.4]$ would fall in the same bucket). The required aggregate over the dependent variable can now be computed by summing the values of definite integrals over all the pieces within each hash bucket, as explained earlier.

Grouping the dependent variable (*e.g.*, Query 5 in Section 2.1) uses the same inverse transformation as discussed for joins. The algorithm first invokes the inversion primitive on each tuple to express x in terms of y, and then follows a split-and-hash procedure identical to that described above to group on the (now independent) variable y.

4.4 Evaluation

In this section, we present an experimental evaluation of the FunctionDB query processor on queries from the applications described in Chapter 2. We first show a simple moving average query on which using FunctionDB to fit a regression model helps deal with gaps in raw data. We then quantify two main advantages of our algebraic query processor over the gridding approach used by systems like MauveDB, where models are represented as discrete points by evaluating the regression model at a fixed gridding interval.

First, gridding introduces *discretization error*, because a grid is a coarse approximation to a continuous model. We show that this error can be significant for simple aggregate queries on our data. In contrast, FunctionDB queries avoid gridding models until the output stage of a query plan, and hence do not introduce any additional discretization error.

Second, while it is sometimes possible to reduce discretization error by using a small gridding interval, this requires storing and/or processing a large number of discrete points. FunctionDB's algebraic approach is a performance win, because it only has to process as many tuples as there are pieces in the regression model. Although algebraic manipulations on tuples are slightly more complex than relational operations on individual raw data points, our results demonstrate that in practice, this tradeoff largely favours algebraic query processing. The lower footprint of our approach results in reduced CPU cost for per-tuple processing overhead, memory allocation and deallocation, and substantially lower I/O cost.

Finally, we also present a cross validation experiment from the Cartel application (Section 2.2) that demonstrates the benefits of using FunctionDB to smooth over gaps in raw GPS data.

4.4.1 Experimental Methodology

For evaluation, we built an in-memory database prototype of FunctionDB in C++. We constructed FunctionDB query plans by hand, by connecting together FunctionDB operators. For each experiment, we also implemented a gridding version of the query which operates on a model representation gridded at regular intervals of the independent variable, and uses traditional query processing operators. In all our experiments (algebraic and gridding), the query processor reads data stored on disk into an in-memory table, executes the query plan on the in-memory table, and writes the query results to a file on disk. We have chosen to build an in-memory prototype of our query processor for simplicity, because our data sets fit in main memory. Our experiments quantify I/O cost (reading data from disk) separately from CPU cost (query processing). The CPU cost measures query execution time when all the data is already in the DBMS buffer pool, while the I/O cost provides insight into how performance might scale to larger on-disk datasets. All our experiments were run on a

3.2 GHz Pentium 4 single processor machine with 1 GB RAM and 512KB L2 cache. Our results are all averaged over 10 experimental runs.

4.4.2 Part A: Temperature Sensor Application

We evaluated FunctionDB on the temperature data described in Section 2.1. To recap, this data contains temperature observations collected from 54 sensors, with the schema <`time`, `temp`>. We first fitted a piecewise linear regression model to temperature using a peak finding procedure, as described in Section 2.1. For testing, we inserted all the regression models into a single function table, `tempmodel`, containing 5360 function pieces. This corresponds to 10 days of temperature data, with a total of ~ 1,000,000 temperature readings. Each piece describes `temp` as a linear function of `time`, and on average fits approximately 200 raw temperature readings.

Comparison to Raw Data

We present a simple comparison of FunctionDB to query processing over raw temperature data. Figure 4-1 shows the results of a simple query that computes moving averages of temperature over 10 second windows (Query 4, Section 2.1). The figure shows that computing the average over raw data is inaccurate and yields spurious results whenever there is a gap in the data. Running the query over regression functions yields a smoother moving average without outliers.

Comparison to Gridding

To compare to a gridded representation of the model, we evaluated FunctionDB on the histogram query from Chapter 2, Section 2.1 (Query 5). The query computes a histogram of temperatures over the time period of the dataset, using temperature bins of width B_0 (a parameter). For each bin, the height of the histogram measures the total length of time (summed over all sensors) for which any of the locations experiences a temperature that lies in the range specified by that bin. This query involves a grouping operation prior to aggregation:

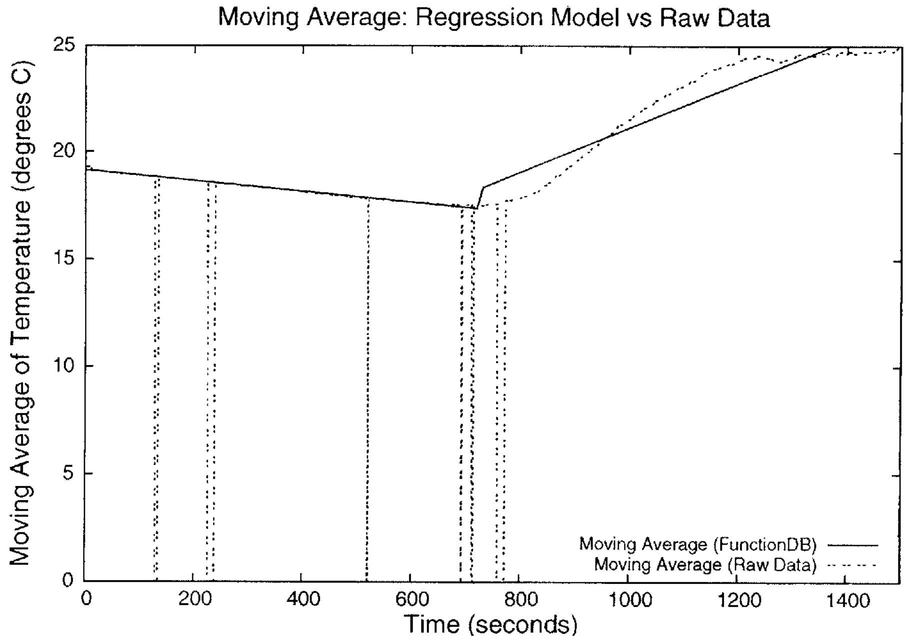


Figure 4-1: 10-second moving averages over 25 minutes of temperature data, computed over raw data and over a piecewise linear regression model with FunctionDB. The spikes in the average are errors due to missing data, which are corrected by regression.

```
SELECT AMOUNT(time) FROM tempmodel
GROUP BY temp GROUPSIZE B0
```

The FunctionDB plan for the above query consists of an algebraic GROUP BY operation on the dependent variable, temp, followed by an AMOUNT aggregate over the independent variable, time. The query consists entirely of algebraic operators. The GROUP BY operation works by first inverting the function and then splitting temperature intervals into groups with the specified size B_0, and the aggregation works by summing the lengths of all the time intervals that fall within each temperature bucket (Section 4.3). The gridding query plan, on the other hand, reads a gridded representation (gridded on time) of data off disk and processes it. This query plan uses a traditional GROUP BY operator to map temperature values to bins of size B_0, and the discrete SQL COUNT aggregate to count the number of time samples that lie within each bin. The count of time samples within each temperature bin is used to approximate the actual query result.

Figure 4-2 shows the execution time for the histogram query when using FunctionDB, as compared to gridding. Results are shown for 4 values of the grid size at which the query

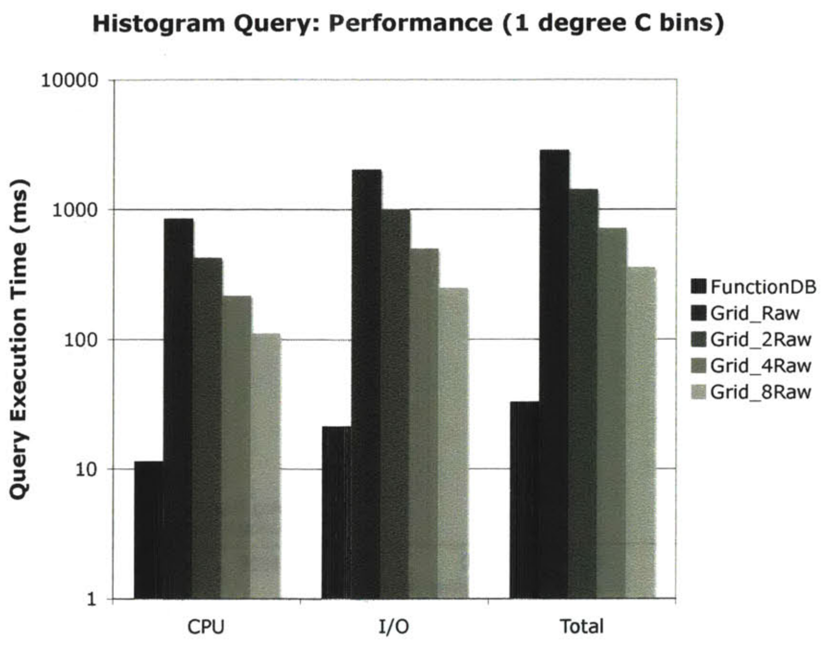


Figure 4-2: Performance of FunctionDB compared to gridding with different grid sizes, on histogram query. The base grid size, "Grid_Raw", is equal to the average spacing of raw temperatures (1 sec). The other three grid sizes are successively larger multiples of this spacing (2 sec, 4 sec, 8 sec).

can be executed without significant loss of accuracy. The figure shows that FunctionDB is faster than the gridding strategies by an order of magnitude in terms of both CPU and I/O cost. Both CPU and I/O savings are due to FunctionDB's small footprint: the FunctionDB query plan needs to read, allocate memory for and process only 5360 tuples (one per function piece). On the other hand, "Grid_Raw" (gridding with the same spacing as the raw data) needs to read and process $\sim$ 1,000,000 discrete points, and hence performs an order of magnitude worse.

Figure 4-2 indicates that it is possible to reduce gridding footprint, and hence processing time, by widening grid size. Doing so, however, adversely impacts query accuracy. Figure 4-3 shows the discretization error introduced by gridding (averaged over all bins) on the histogram query, as a function of grid size. The error is computed as a percent deviation from the result of the algebraic query plan, which does not suffer from this error. The graph shows that discretization error grows significantly with grid size. Hence, using a widely spaced grid is not a viable option.

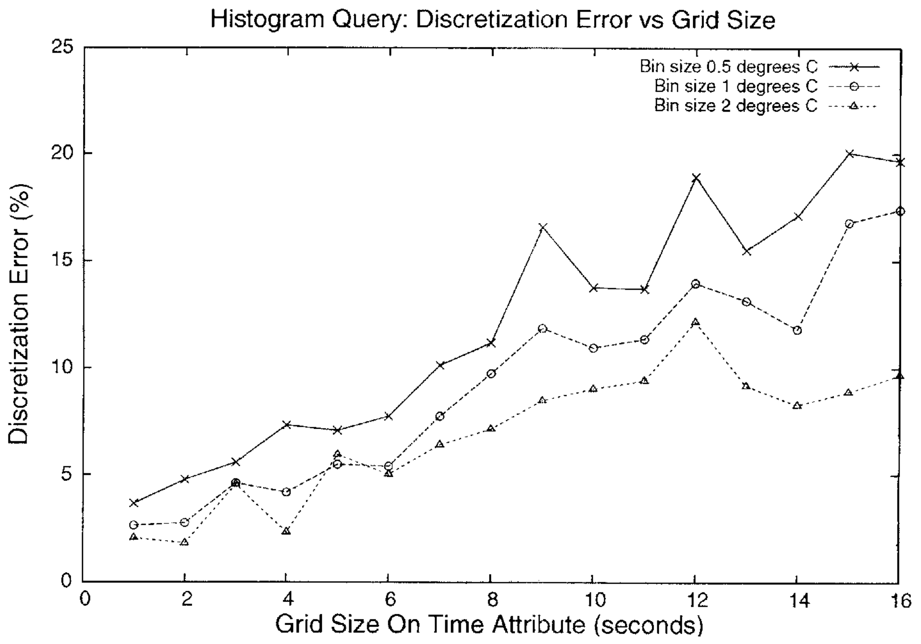


Figure 4-3: Discretization error (% deviation from algebraic answer) due to gridding, as a function of grid size, averaged over all bins for the histogram query.

The discretization error in Figure 4-2 results from sampling, which is an imperfect way to compute the aggregate of a continuous function. The gridding approach samples temperature data at discrete time intervals and counts the number of temperature observations in each bin. This becomes inaccurate when the sampling interval is comparable to a time scale over which temperature varies significantly (and hence affects the bin to which sampled temperature readings are assigned). As the figure shows, the error is exacerbated for queries that group temperature into smaller bins.

While we have used FunctionDB results as the baseline for computing error in the above experiment, the actual accuracy of a FunctionDB query plan is clearly limited by the inherent error in raw data and in the model fit by the user. The experiments above (and for the trajectory similarity query, which follows) are mainly intended to show that gridding introduces significant *additional* error, while FunctionDB does not. An end-to-end comparison of query accuracy is difficult to obtain in this application, because the sensor data is noisy in addition to having missing values, and hence there we do not have a ground truth to compare against. On the other hand, the GPS trajectory data (Section 4.4.3, which follows this section) is more accurate, with gaps in the data being the chief rationale for modeling (with the exception of a few outliers which are possible to recognize and filter out after performing the regression fit). It *is* therefore possible to validate our regression fit in that example: the results of a systematic cross-validation experiment for the Cartel trajectory data are presented in Section 4.4.4.

Also, we recognize that it might be possible to implement an adaptively spaced sampling strategy to reduce discretization error, but this is more complex to implement and harder to generalize to arbitrary queries. Algebraic query processing permits a simpler implementation, achieves zero discretization error, and at the same time provides good performance.

While algebraic query processing results in a clear performance win for aggregate queries that do not need to grid results at any point in a query plan, some queries (including simple selections) do need to grid results for display or writing to a file, in order to provide the familiar output semantics of a traditional DBMS. In such situations, algebraic query execution does not always outperform gridding, but is still beneficial for *selective* queries, if gridding is performed after selection. Our second query determines when the temperature exceeds a particular cutoff, T_0:

```
SELECT time, temp FROM tempmodel
WHERE temp > T0 GRID G0
```

The query takes an additional parameter G_0, which specifies the granularity with which results are gridded for display to the user. The FunctionDB query plan for selection uses an algebraic root finding procedure (Algorithm 1, Section 4.3), but then applies `GRID` to the selected functions for displaying the result of selection. The gridding approach applies a traditional selection directly to data gridded at regular intervals of the `time` attribute.

Figure 4-4 shows the total query execution time (CPU + IO) for the above query, for three different values of the cutoff temperature T_0. These values correspond to three different predicate selectivities: Not Selective ($\sim$ 70% of tuples pass the filter), Selective ($\sim$ 20% of tuples pass), and Highly Selective (only $\sim$ 2% of tuples pass). Algebraic query processing yields significant benefits for the selective query plan, because this plan only needs to grid 2% of the data for output. The benefits are less pronounced for the less selective query plan, and there is no difference for a query plan that needs to output most of the data, because gridding is the main bottleneck in both query plans. We have repeated this experiment for larger values of output grid size G_0 ; the relative results and conclusions are unchanged as long as the grid size is not too wide.

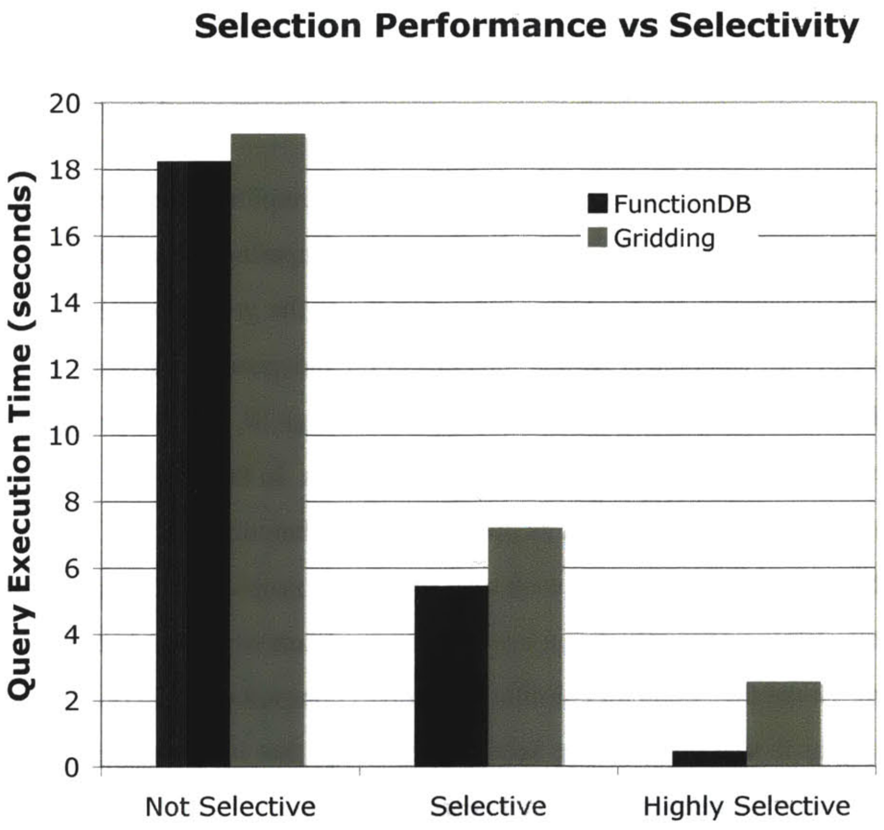


Figure 4-4: Selection performance of FunctionDB compared to gridding with output grid size 1 second, for 3 values of selectivity: 70%, 20% and 2%. Algebraic processing is a win for selective queries, which do not grid much data for display.

4.4.3 Part B: Car Trajectory Application

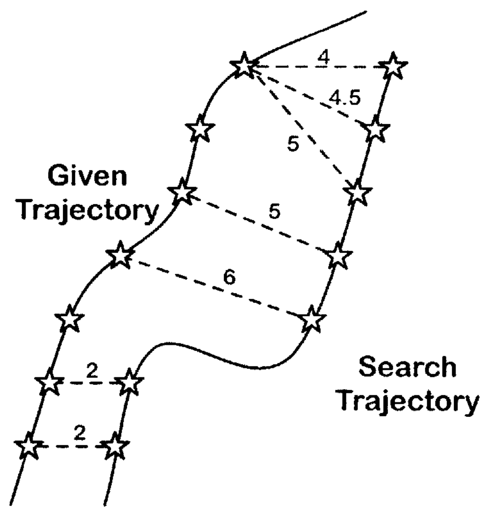
Our second evaluation dataset consists of vehicle trajectories from Cartel [8], fitted with a piecewise linear model for road segments. The data consists of 1,974 vehicle trajectories collected over a year. Our regression model consists of 72,348 road segment "pieces", used to fit 1.65 million raw GPS readings. This model fits fewer raw data points per piece (~ 20) than the regression model for temperature, because trajectories are sometimes curved or include many turns, requiring multiple line segments for approximation.

For evaluation, we used a variant of the trajectory similarity query described in Section 2.2. Given a trajectory identifier, this query finds trajectories whose endpoints are within 1' of latitude and longitude (corresponding to $\sim$ 1.4 km) of the endpoints of the given trajectory. For each such neighbouring trajectory, the query lines up points on the given trajectory with points on its neighbour based on a "distance fraction" criterion (Section 2.2), and computes the average distance between pairs of lined up points as a similarity metric between the trajectory and its neighbour.

To focus our benchmark on join performance, we simplified the query somewhat for both the FunctionDB and gridding approaches, by precomputing a materialized view, `fracview`, with the logical schema <`tid, lon, lat, frac`>. In the gridded representation, points in the trajectory are sampled at regular intervals of the independent variable, `lon`. For each point in the trajectory, `frac` is a real number in the range $[0, 1]$ representing the fraction of distance along the trajectory at which the point occurs. In the function table representation, both `lat` *and* `frac` are stored as piecewise linear functions of `lon`. This is possible because the increase in `frac` along each road segment is proportional to the distance along the segment, $\sqrt{lat^2 + lon^2}$, which is a linear function of `lon` whenever `lat` is a linear function of `lon`. Given this, trajectory similarity involves computing a join on `frac`. The SQL query is similar to that presented in Section 2.2, except that it does not compute all pairs similarities, and it includes an additional nearness criterion.

The FunctionDB plan for trajectory similarity is completely algebraic. The major step is an NL-Join on `frac` (the dependent variable) using function inversion (Section 4.3). Joined trajectories are grouped on `tid` using a traditional `GROUP BY`. The last step maps the algebraic expression for Euclidean distance to the functions in each tuple, and computes an `AVG` aggregate over this expression. The expression is a hyperbolic function of `lon`, of the form $y = \sqrt{ax^2 + bx + c}$. We have manually added this type by implementing algebraic primitives for it. As described earlier in this chapter, arithmetic expression types can usually be inferred at compile time in the FunctionDB framework.

The procedure for the gridded version of the join on `frac` is illustrated in Figure 4-5. Each grid point on a search trajectory is matched to the point with the closest value of `frac` on the (given) query trajectory using binary search on a tree data structure built on the fly,



Total similarity:AVG(2,2,6,5,5,4.5,5)=4.1

Figure 4-5: Lining up two trajectories using a grid approach.

and average distance is computed between the pairs of points lined up in this way.

Figure 4-6 shows a CDF of the discretization error in the similarity metric (averaged over search trajectories) computed by the gridding approach. The distribution is computed over different choices for the query trajectory, and is shown for two grid sizes, one equal to the raw data, and the other 8 times this size (the curves for 2 and 4 times lie in between). The graph shows that median discretization error is larger for the wider grid size. Also, for both grid sizes, portions of the distribution experience significant discretization error (> 15%).

Discretization error occurs in this example because the data is gridded along the `lon` attribute (Figure 4-5). Hence, each trajectory has an unequal number of grid points, unevenly spaced along the join attribute `frac`, resulting in imperfect pairings. While using a representation gridded over `frac` could be more accurate in this particular case, this might not work well for other queries. It may be possible to maintain multiple gridded representations and infer which one to use for each query, but this would add complexity and redundancy, and would be harder to maintain.

Figure 4-7 shows the corresponding performance results, averaged over query trajectories. FunctionDB outperforms all but one of the gridding strategies, and at the same time has no discretization error. The improvements are not as dramatic as in the temperature

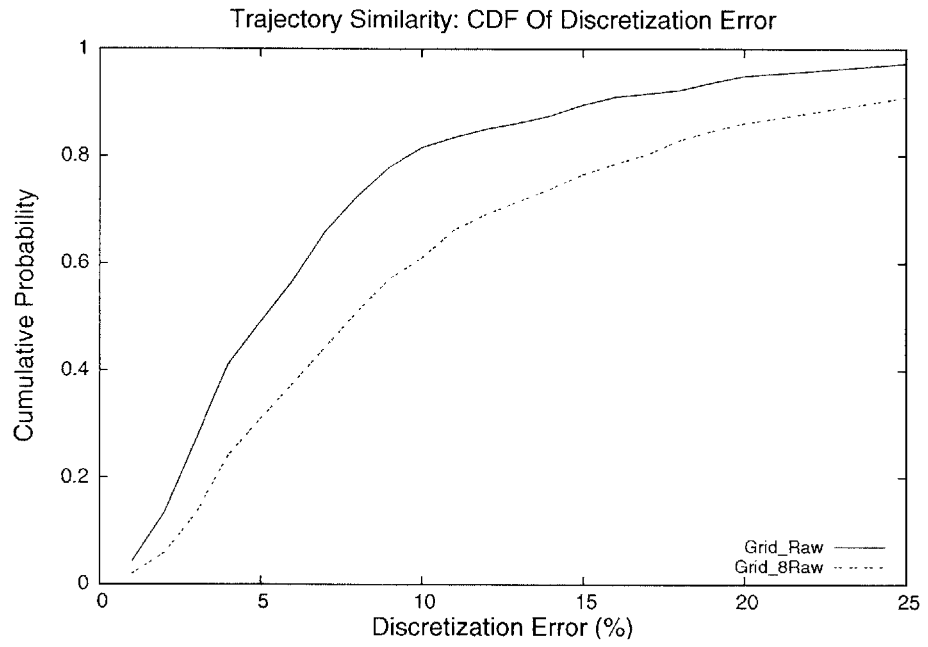


Figure 4-6: CDF of discretization error for two different grid sizes on trajectory similarity query, computed over query trajectories. The error distribution has a significant tail.

histogram, because the ratio of number of grid points to pieces is somewhat lower than in that application (as mentioned earlier). However, these results do show that algebraic processing has significantly better performance than discrete sampling.

4.4.4 Part C: Validating Model Quality

As mentioned in the discussion of accuracy results, it is difficult to estimate end-to-end query accuracy in experiments without knowledge of the ground truth, especially when the raw data is itself noisy. A workaround to this problem, typically used in machine learning, is *cross validation*. The high-level idea behind cross validation is to divide the available data into two sets: a training set and an unknown test set. A model trained on the training data is evaluated on the test data in order to gauge how effectively it can predict the ground truth.

Two popularly used forms of cross validation are leave-one-out cross validation (LOOCV) and k-fold cross validation. We use the latter approach. In k-fold cross validation, the data is divided into k subsets and cross validation includes k steps. Each step involves leaving out each of the k subsets in turn, training on the remaining (k-1) sets, and making predic-

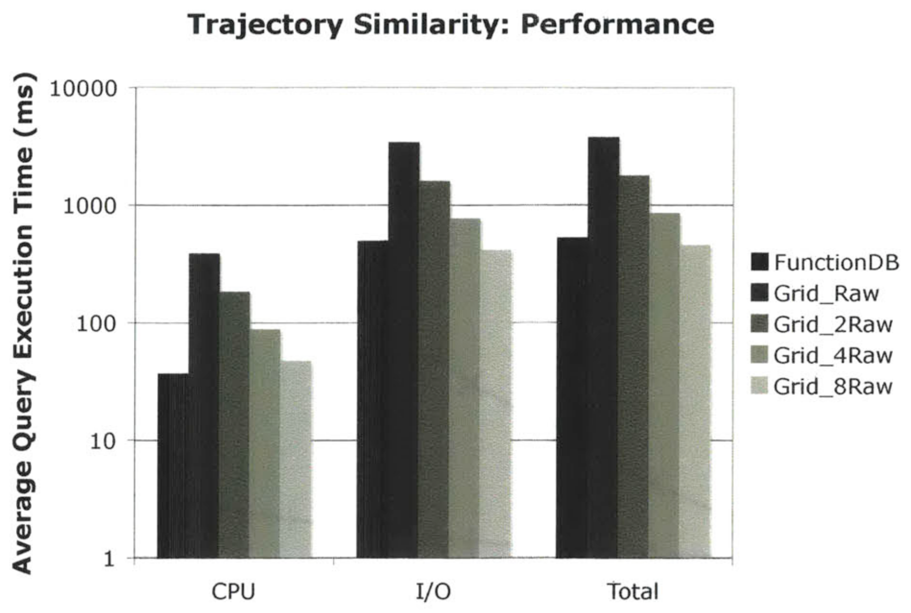


Figure 4-7: Performance of FunctionDB on the trajectory similarity query, compared to gridding with different grid sizes. As before, the grid sizes are multiples (1x, 2x, 4x, 8x) of the average spacing of raw data in the data set.

tions on the kth set which was left out. The error is averaged over all choices of the set left out. It has been shown that k-fold cross validation produces an unbiased estimate of the model generalization error.

It is possible to do cross-validation for our trajectory data by choosing contiguous segments of GPS readings which are known not to be outliers, and are hence reasonably accurate. We now inject *gaps* into this data by leaving out consecutive segments of raw data observations. The available raw data constitute the training set for a regression model, and the missing (injected) gap constitutes the test set. The model can be validated by comparing its predictions for values in the injected gap with the known values which were left out, and measuring the model prediction error, given by the deviation of model prediction from the true value. The process is repeated for different positions of the gap (as in k-fold CV) in each trajectory, and prediction error is averaged over all these positions.

Figure 4-8 shows the result of cross validation as applied to a subset of the Cartel trajectory data, for the regression functions used in the experiments presented earlier (these were generated using an update algorithm which will be described in Chapter 5). Because cross validation is expensive, we used a randomly selected subset of the data (consisting

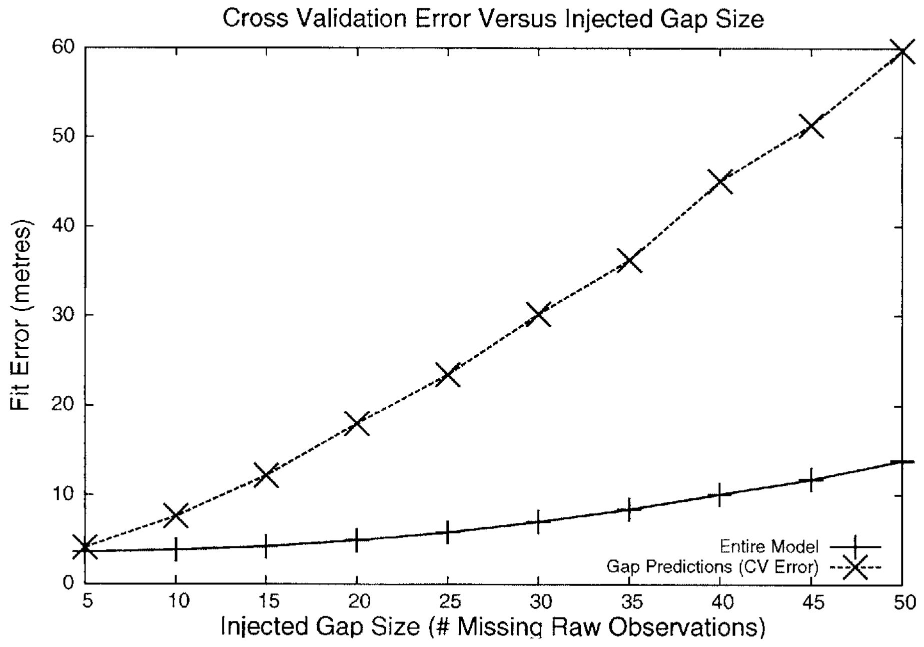


Figure 4-8: Cross Validation (Gap Prediction) Error and Total Error, as a function of the number of missing raw data observations.

of 40 trajectories) to conduct the experiment. The plot in Figure 4-8 shows the average prediction error for points in the gap, as well as over the entire trajectory, as a function of the number of observations left out for cross validation. For large gap sizes (> 30 missing observations, corresponding to nearly a minute of missing GPS readings), the average CV error is much larger than the GPS precision, but for smaller gaps less than a minute's worth of missing data, the plot shows that regression is quite effective in predicting the missing data.

Most of the cross-validation (CV) error in the above experiment results from missing data at road intersections and turns (*i.e.*, boundaries between successive model segments), which is understandable. An underlying model of the trajectories based on actual road segment data as opposed to regression would work better.

Chapter 5

Updates

Chapter 4 illustrated the benefits of algebraic query processing after regression functions have already been fit to the data in question. In practice, however, data is not static. There is a performance cost to fitting and maintaining a regression model as new data is inserted into FunctionDB.

This chapter discusses the implementation of data updates in FunctionDB, focusing on the case of appending high-rate streaming raw data into the database. This is as opposed to handling in-place "point" updates to data, where refitting the entire regression model is unavoidable if the existing regression model does not approximate the new data well enough.

We introduce and discuss the major design considerations for online model maintenance algorithms, and use these to guide the development of an online search-based algorithm, `BSearch`, for approximating one dimensional streaming time series data with a set of regression functions. The algorithm is extensible to piecewise functions fit using linear regression (as defined in Chapter 2, Section 2.1): the basis functions for regression can be non-linear. In addition, the algorithm has a sufficiently flexible design to incorporate a variety of accuracy criteria useful in practice, like model generalization performance.

There is a large body of previous work on segmenting time series data (see, for example, [11] for a comparative survey): the purpose of this chapter is not to claim fundamentally new algorithms for this task (our algorithm draws liberally on previous work), but to implement and evaluate a practical, generalizable algorithm in the context of FunctionDB. We also justify specific advantages of our choice of algorithm compared to previously proposed solutions, in the context of our design goals.

We present an evaluation of `BSearch` on streaming car trajectory data from Cartel, and show that it can support a peak (in-memory) insertion rate of nearly 1 million records/second for piecewise linear regression on the Cartel application. At the same time, it generates models with reasonable accuracy of fit which are compact (measured in terms of the number of pieces in the output regression model) compared to a clairvoyant (offline) model fitting strategy that knows all the insertion data in advance.

5.1 Design Goals

We consider the problem of maintaining a regression model on an ordered stream of data with schema `<x,y>`. Since the task of fitting a single regression function to a block of time series data is easy to accomplish with Gaussian elimination (Chapter 2, Section 2.1), the main design challenges are: (a) determining the best places to cut the data into blocks or "pieces", so that a single model is fit to the data within a piece, and (b) deciding how much historical data to retain and use for fitting. All the data is not be available in advance, and an algorithm cannot keep an arbitrary large window of data history (because it would be too expensive). With the above points in mind, we detail the important design goals for an update algorithm in the sections that follow.

5.1.1 Model Quality

Model quality has two dimensions: accuracy of fit, and compactness, which affect query accuracy and performance, as we have seen in Chapter 4.

Fit Accuracy. The most important consideration is how well the model serves as an approximation to ground truth.

In situations where the raw data are reasonably free of measurement noise, and modeling mainly serves to detect outliers or fill gaps in the data (*e.g.*, GPS data from car trajectories), fit accuracy is easy to measure using a standard fit error metric, like the RMS (Root

Mean Squared) error of fit, given by $\sqrt{\frac{1}{N}\Sigma_{i=1}^{N}(y_i - M(x_i))^2}$, where (x_i, y_i) are the raw data observations (numbering N) and $M(x)$ is the value of y predicted by the regression model for a particular value of x.

In situations where the raw data has measurement noise and there is no easy way to verify against ground truth, cross validation (as described in the previous chapter) can be used to gauge the generalization performance of a regression model, and ensure against overfitting the data. Here, the cross-validation (CV) error measured using k-fold or leave-one-out cross validation is an appropriate metric to use in place of the raw fit error.

Model Compactness. The second measure of model quality is the compactness of the output model. As shown in the query processor evaluation presented in the previous chapter, compactness has a direct bearing on query processing performance: a more compact model is substantially faster to process in terms of both CPU and I/O cost. In the context of online model maintenance, compactness is challenging to achieve, as we shall see. This is because generating a compact model requires maintaining a history of raw data tuples as well as the model pieces already fit to this window of history, to avoid unnecessarily creating a new piece every time a new tuple (or a small number of new tuples) arrive(s) into the system. Model compactness can also be at conflict with the previously mentioned objective of fit accuracy, because using fewer pieces to approximate the data requires compromising on the fit error for some points, especially if the data is only imperfectly described by the class of curve being used for fitting.

5.1.2 Update Performance

The second important consideration is update performance *i.e.*, how fast the system can update an existing regression model when new data comes in, and whether the system can keep up when new data is inserted at a high rate. As in the case of model quality, performance has two (related) dimensions:

- **Throughput**, which is the maximum rate at which the system can insert new data into the database and build (or rebuild) a regression model to cover the newly inserted data.

- **Latency**, which is the time delay from arrival of a new raw data observation to when it is reflected in the function table representing the regression model.

5.2 The `BSearch` algorithm

One straightforward strategy to break data into pieces would be to use an algorithm like `FindPeaks` (mentioned in Chapter 2), which finds likely breakpoints in the data. Unfortunately, this algorithm (finding peak or valley extrema in data) is quite specific to linear functions, and difficult or impossible to generalize to nonlinear models. Further, it does not provide a straightforward way to tune the accuracy of the generated fit or trade this off for a more compact model.

For these reasons, we seek an algorithm which provides the user with direct control over the accuracy of the generated fit. The next section introduces and describes a search-based algorithm, `BSearch`, which does provide this control. The focus is mainly on data that can have gaps, but has little or no measurement noise (like GPS data). Hence, the RMS error of fit (as opposed to a generalization error metric, like cross-validation error) will be the primary accuracy criterion used in the discussion and in the subsequent evaluation. The algorithm itself should generalize to any error metric.

The high-level idea behind the segmentation algorithm described in this section, `BSearch` is quite simple. The algorithm essentially works by dividing a block of data into pieces in a top-down fashion until the models fit to *all* the resulting pieces satisfy a user-specified threshold on the RMS error of fit (or other accuracy criterion), which constitutes a parameter to the algorithm. While `BSearch` shares the same high-level idea with several offline algorithms previously proposed, for example, in the context of image processing, graphics and time series data mining [10, 11, 23], it differs in some important details.

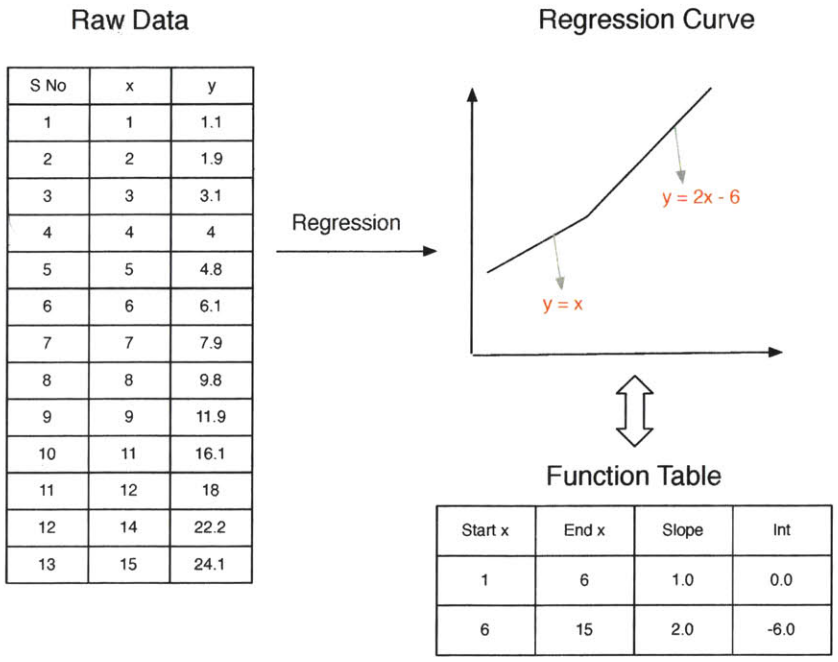
Adapting the above idea from an offline algorithm on a block of data (as described above) to an online algorithm on streaming data requires maintaining a window of historical data, say W, and a pointer to the models that have previously been fit to data in this window, if any (say M_W). `BSearch` works within the block of data specified by W and tries to find appropriate points to divide the data into pieces, as per the user-specified error threshold.

The top down algorithms discussed in previous works use a simple divide and conquer strategy by picking the best point in W to split the data into two parts, and recursively invoking the algorithm on each part if the fit does not satisfy the error criterion. `BSearch` chooses to instead use binary search to iteratively find pieces satisfying the error criterion that are as large as possible. We will shortly show (Figure 5-5) that in conjunction with batch updates, our search strategy enables us to generate more compact models for the same error threshold compared to a simple top-down strategy.

More precisely, the `BSearch` algorithm works on W from left to right, starting a new piece at the left endpoint of the block. The first decision to make is where to end this left-most piece. The search proceeds as follows: `BSearch` considers the right endpoint of W as the first candidate endpoint, and uses linear regression (with the given basis functions) to test if this will satisfy the user-specified constraint on RMS error. If the piece has RMS fit error larger than the RMS threshold, the midpoint of W is chosen as the next candidate right endpoint, and so on in a fashion reminiscent of binary search — until the first end-point satisfying the error constraint is located. The search does not stop here: rather, this discovery narrows the search for the rightmost endpoint.

To illustrate, we revisit an example similar to that used in Chapter 3. This example is shown in Figure 5-1. A sequence number has been added to the raw data table for convenience of illustration. Assuming the 13 raw data readings shown in the table form a window, `BSearch` would first consider fitting a model to the entire range of data *i.e.*, from sequence numbers 1 to 13 (inclusive). In this case, it turns out that this model has unacceptably high fit error, so the algorithm tries $\frac{1+13}{2} = 7$ as the next candidate endpoint. Again, fitting data from sequence numbers 1 to 7 fails to make the cut, so $\frac{1+7}{2} = 4$ next is tried next. Since the data in [1..4] satisfies the error criterion, the search for the best endpoint narrows down to $[4, 6]$. The binary search terminates with the data in $[1, 6]$ being recognized as the largest piece satisfying the error criterion. The function fit to this data ($y = x$, as shown in the figure) becomes the leftmost piece of the overall regression model.

Having identified the leftmost piece, the left endpoint for search is now advanced beyond this piece (to 7 in the above example) and the procedure is repeated to locate the endpoint for the next piece. The process terminates when the last such piece is fit to the



S No	x	y
1	1	1.1
2	2	1.9
3	3	3.1
4	4	4
5	5	4.8
6	6	6.1
7	7	7.9
8	8	9.8
9	9	11.9
10	11	16.1
11	12	18
12	14	22.2
13	15	24.1

Start x	End x	Slope	Int
1	6	1.0	0.0
6	15	2.0	-6.0

Figure 5-1: Example raw data and corresponding function table.

data in *W* (this happens when the identified breakpoint coincides with the right endpoint of *W*). The procedure is formally presented in Algorithm 3.

`BSearch` uses binary search to keep the number of model fits (and hence matrix inversions) within O(M log N) where M is the number of pieces in the output model, and *N* is the size of the window of data *W*. Note that M depends on the chosen error threshold *ET* and the distribution of the raw data in a complex way: while we do not include a theoretical analysis in this thesis, Figure 5-4 (Section 5.3) presents a practical measurement of this dependence on the vehicle trajectory data set from Cartel.

`BSearch` also uses memoization to save on some computation (the "Basis" array computed in step X). This is possible thanks to a nice property of linear regression: the coefficients of the basis matrix used for regression can be computed for an arbitrary interval as the difference of two cumulative sums: *W*[L..R] = Basis(R) - Basis(L). This is a generalization of the observation made in [7] that an intermediate representation of sums can be used to speed up regression fitting.

Algorithm 3 by itself is not enough to describe an online algorithm. After fitting models to the data in *W* as described above, the next step is to slide the window of history when

Algorithm 3: The BSearch Algorithm

```
Given: A window $W$ with N $(x_i, y_i)$ observations, and an RMS error threshold $ET$.
Basis ← Precompute basis matrix for $W$[1..R] where $R \in 1, 2, .., N$
Left ← 0
SearchL ← 0
while $Left \leq N$ do
    SearchL ← Left + 1
    SearchR ← N
    while $SearchL \leq SearchR$ do
        (Model, RMSError) ← Fit($W$[Left..SearchR]) using Basis
        if $RMSError < ET$ then
            SearchL ← SearchR + 1
            SearchR ← N
        else
            SearchR ← $\frac{SearchL+SearchR}{2}$
    Output model for data in $W$[Left..SearchR]
    Left ← SearchR + 1
```

new tuple(s) arrive. There are two important decisions that need to be made here:

History Size. BSearch needs to decide how much historical data to retain for the next invocation of the search algorithm *i.e.*, the size of the window W referred to above. In this regard, we propose to minimize retained history by only retaining the data corresponding to the *rightmost* piece that was fit to data in the previous window, and "freezing" (fixing) all the previous pieces. The rationale is that retaining more history is practically useless, because the next phase of the search algorithm will invariably find the same breakpoints as the previous phase, and is thus redoing work unnecessarily in most cases.

There are exceptions to the above statement when the sliding window of data is small, or the data is noisy. In these cases, freezing a segment of history without knowing all the data in advance can mean that BSearch is sometimes a little too aggressive in breaking data into pieces, resulting in slightly increased model size. However, as Section 5.3 will show, the cost is quite small: our strategy of retaining only one piece's worth of data, when coupled with a sufficiently large batch size for updates (see below) generates models with size no worse (larger) than the model that would be fit by a clairvoyant strategy that knew all the data in advance.

Batching. BSearch also needs to decide how often to rerun the search algorithm. In other words, do we rerun the search algorithm for each new incoming tuple or not? An alternative we consider and evaluate in this chapter is a "batch update" technique: instead of rerunning the model fitting process for each new incoming tuple, the system waits for a fixed number of new input tuples to accumulate, and invokes Algorithm 3 only when sufficient new tuples are available. A form of batching was first proposed by [11] in the context of a bottom-up segmentation algorithm; here, we implement and evaluate its benefits in the context of our `BSearch` algorithm.

Batching updates potentially increases throughput by reducing the number of invocations to the potentially expensive fitting process, but at the expense of increased latency. Moreover, batching also affects the quality of fit (both accuracy and size) in a non-obvious way because the size of batches seen by the search algorithm affects the regression functions fit to the data. Our evaluation in Section 5.3 strives to measure these tradeoffs, and thus gauge the effectiveness of batch updates.

5.3 Evaluation

We evaluate the `BSearch` algorithm presented above, with respect to both peak achievable update throughput, and quality of fit (in terms of both model size and accuracy), on the Cartel trajectory data described in Chapter 4. Note that the `BSearch` algorithm with batching, as presented above, has two important knobs that govern performance and fit quality: the RMS error threshold ET and the batch size used for updates, which we denote by B. One aim of this experimental study is investigating how to set appropriate values for ET and B, and how data- or application- dependent the best values for these parameters are likely to be. We also compare `BSearch` to a simple divide-and-conquer top-down strategy discussed in previous work ([11]), and show that principled search finds more compact models for a given error threshold.

We measure the impact of batch updates in both the performance and accuracy experiments by varying the batch size B used for updates. The experimental methodology is similar to that used for the query processing experiments in Chapter 4: a file containing raw data is first read into memory. Data from this file is now inserted into a function table in a rate-controlled fashion at a fixed offered rate. The experiment measures the total real (wall clock) time for inserting all the records from the file into the function table, and uses this to calculate the actual achieved insertion throughput. Batch updates are implemented as follows: the FunctionDB update operator waits for a fixed number of records to accumulate before inserting them into the table using one pass of Algorithm 3.

5.3.1 Update Performance

Figure 5-2 shows a plot of insertion throughput in FunctionDB against the offered rate of insertions. The plot is shown for different values of the batch size B. The RMS error threshold parameter ET is kept fixed in all the experiments, at a value of 0.0001^o of longitude ($\sim$ 5.5 metres), which is approximately on par with the precision of the GPS device used in Cartel.

The plot shows that batching is beneficial, and a batch size of $B \sim 100$ records results in the peak update throughput of $\sim$ 900K records/sec. As might be expected, the graph clearly shows that a small batch size ($B < 10$) is detrimental to throughput. At the extreme, $B = 1$, which corresponds to rerunning Algorithm 3 for every new record inserted into the function table, achieves a peak insertion throughput of only $\sim$ 110K records per second, nearly 8x worse than the best achieved peak throughput at larger values of B. This clearly shows that choosing B too small is a bad idea.

What might be less obvious is that beyond a certain threshold (between 10 and 100 records in our experiment), batching has diminishing returns or can even hurt performance slightly. The reason for this trend is that beyond a certain threshold, irrespective of batch size, `BSearch` needs to perform computational work proportional to O(M log N), when there are N records in the input data and M is a minimum number of pieces the data divides into, depending on the error threshold ET. For very small batch sizes, `BSearch` generates more pieces than necessary because it keeps a limited window of history, and hence does more work than necessary. For batch sizes significantly larger than a threshold, we therefore see diminishing returns on throughput. This threshold depends on ET, and is close to the number of raw data points that would be fit by each piece when using the best offline

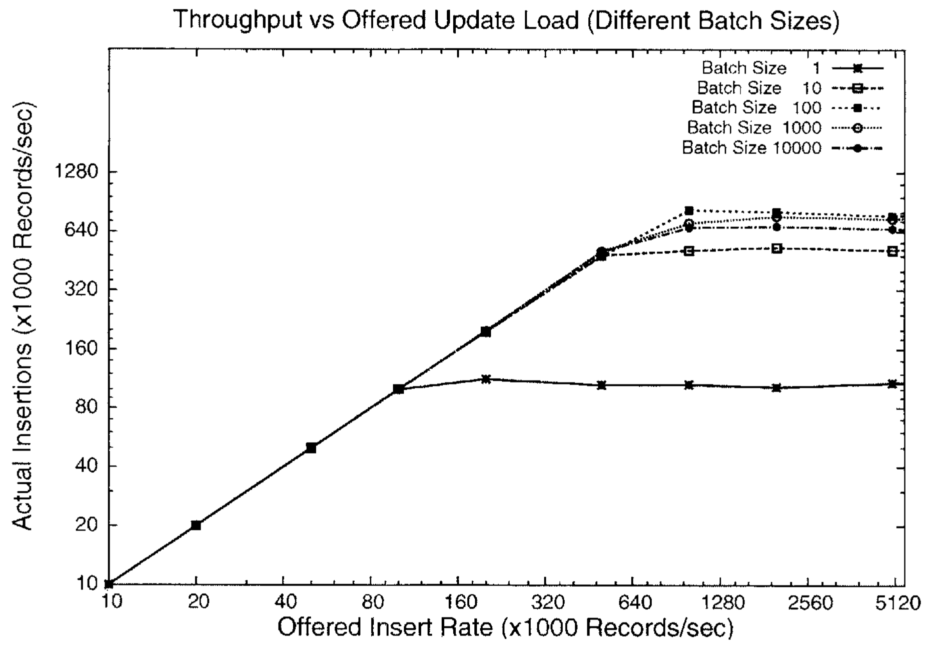


Figure 5-2: Performance benefit due to batch updates.

(clairvoyant) strategy. In our case, this works out to approximately $B \geq 25$ for $ET \sim 5.5$m.

The observation of diminishing returns on throughput is important to keep in mind, because increasing the batch size used for updates has a negative impact on *update latency* *i.e.*, the time from insertion of a record to when this insertion is reflected in the function table. A larger batch size implies a longer wait before insertion, because the system needs to wait until enough tuples accumulate to form a new batch. For a low offered rate (*e.g.*, one record per second) this can mean a long wait. Furthermore, a larger batch size is simply not feasible if the insertions are bursty — *e.g.*, a burst of insertions smaller than the batch size followed by no insertions for a relatively long period of time would mean the original burst would simply not be reflected in the function table, which is unacceptable.

One possible solution to this problem is to *stage* all updates via an intermediate buffer which stores a crude, quickly computed approximation to the latest segments of the regression model (*e.g.*, simply the raw points connected by line segments) which can be queried as an interim solution. Whenever a CPU(s) becomes available, a "cleanup thread" runs over the latest segments of the function table with a larger window size (*e.g.*, > 100) and refits the model using Algorithm 3, updating it in place. We leave implementing and evaluating

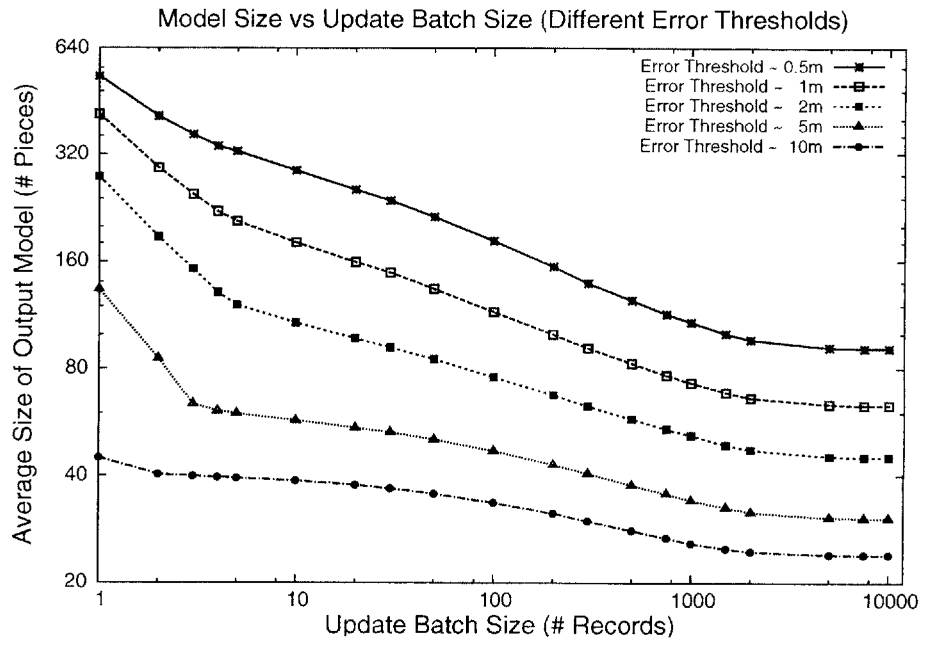


Figure 5-3: Model size penalty due to limited history, for different batch sizes.

a staged strategy based on `BSearch` to future work.

5.3.2 Fit Quality

We first investigate the impact of batch updates on model compactness. Figure 5-3 plots the size of models output by `BSearch`, averaged over all the trajectories in our dataset, as a function of the batch size used in updates. The plot is shown for different values of the RMS error threshold ET. The graph shows that model size does depend significantly on the batch size used for updates — a larger batch size results, on average, in a regression model that is significantly more compact. However, as the graph shows, the output model size does not blow up dramatically until batch sizes smaller than 10 records are used. In particular, the regression model for $B = 10$ is on average only double the size that for $B = 10000$. This result lends support to the thesis put forth in the previous section, namely that increasing B drastically beyond a certain threshold may not be worth the increased penalty in terms of latency.

Figure 5-4 drills down into more detail on the relationship between model size and the error threshold parameter ET when using the `BSearch` algorithm. The plot is slightly pes-

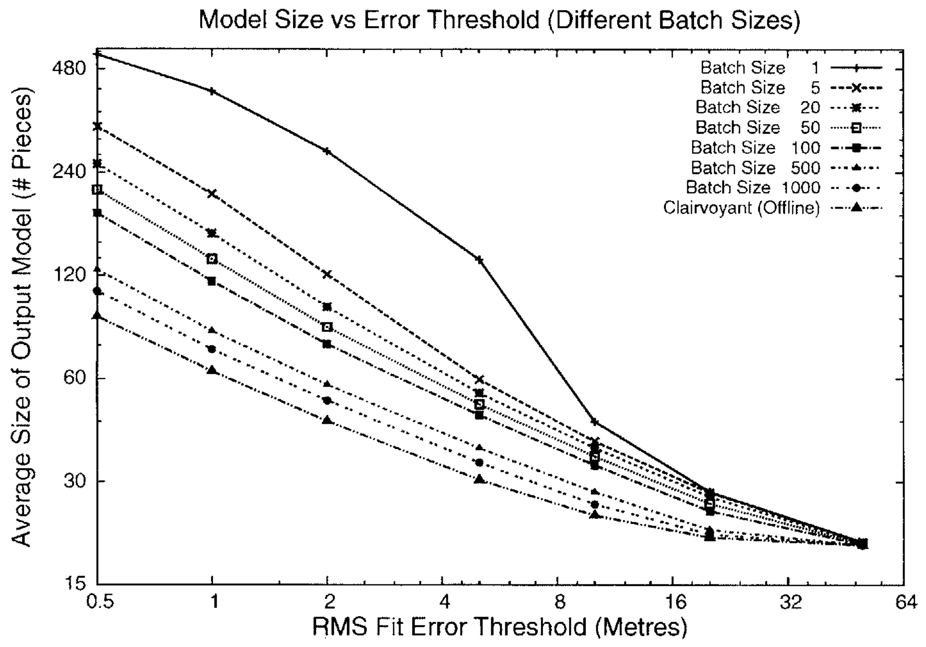


Figure 5-4: Dependence of model size on error threshold.

simistic, indicating a non-linear increase in model size with decreasing error threshold (the graph is on a log scale). However, for the trajectory data, the tradeoff is clearly favourable to functional processing when using a batch size of $B \geq 500$. For example, the query processing performance results presented in Chapter 4 were obtained with an average model size of ~ 40, corresponding to the point ($B = 500$, $ET = 5$m) on the graph, which is a reasonable choice (because $ET \sim 5$m is close to the precision of the GPS receiver). The graph does serve as a word of caution that functional processing may not always be favourable (in terms of performance) for data that is less well described by functions, or requires greater precision.

5.3.3 Comparison to TopDown Algorithm

The advantage of using principled search to find the rightmost position to break off a new model piece is that our BSearch algorithm tends to generate the largest possible pieces given a specific error threshold. This in turn results in a compact model and better query performance. While previous work has considered top-down algorithms which work in a divide-and-conquer fashion (as described earlier), these algorithms tend to be more aggres-

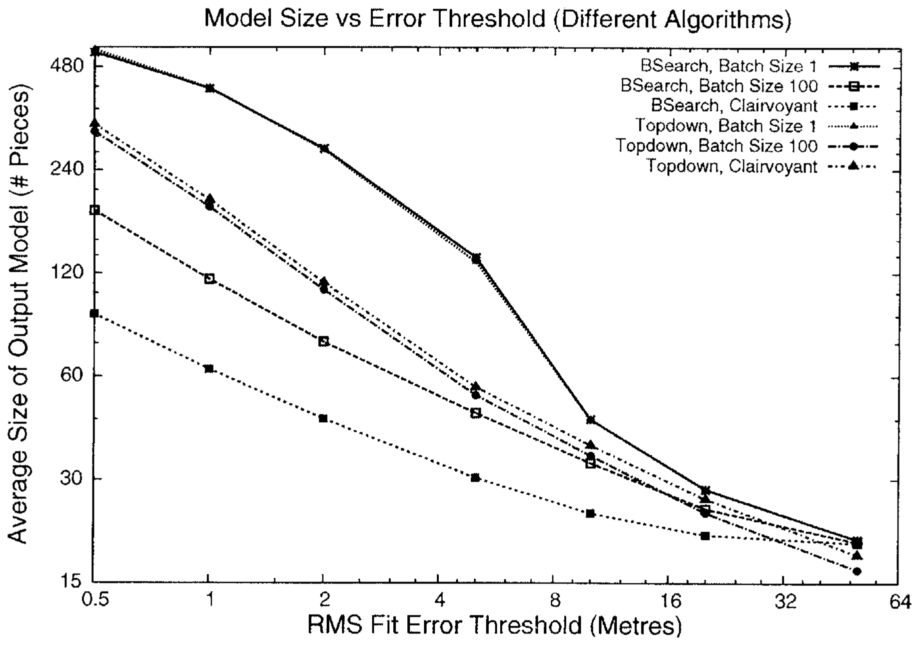


Figure 5-5: Size of models generated by BSearch compared to a simple top-down segmentation algorithm, TopDown.

sive about segmenting data into pieces. A top-down divide and conquer algorithm fits only two pieces to all the data in the window in the first iteration, which can often be insufficient when the window size is significant. For this reason, the resulting models are usually not accurate irrespective of where the algorithm decides to place the cut, and suboptimal information is used to make the splitting decision, often resulting in unnecessary extra splits.

Figure 5-5 demonstrates this experimentally. The plot shows the size of models generated by a simple top-down algorithm, TopDown, as well as by BSearch, plotted as a function of the error threshold. The results are shown for three different batch sizes: 1 (corresponding to no batching), ∞ (corresponding to an offline, clairvoyant algorithm), and 100 (an intermediate batch size). When using a small batch size, both algorithms perform quite poorly in terms of compactness, but as the batch size increases, BSearch outperforms the top-down strategy, and wins by a factor of 2x or more when working offline (corresponding to a batch size of ∞).

5.3.4 Tuning Updates in FunctionDB

Armed with evidence from our performance and accuracy experiments, we now briefly discuss how to set the parameters for `BSearch`, namely B and ET, for different application requirements/contexts.

Setting Error Threshold. Setting the error threshold ET, as we have seen, primarily depends on the relative importance of query accuracy and query performance. However, the exact nature of the tradeoff between performance and accuracy (such as that depicted in Figure 5-4) depends on how well the model fits the raw data. The tradeoff curve itself can be a useful tool for users to pick a value of ET, in conjunction with known values about the measurement precision of the raw data (for example, there is little point in picking ET below measurement precision).

Setting Batch Size. Setting the batch size automatically is more important. Here again, the choice depends on application requirements. For cases where query performance is more important than update latency *e.g.*, expensive queries over less frequently updated data, such as analysis queries over historical time series, a larger batch size makes more sense. For applications where query performance is less crucial, but low latency is a must, a lower batch size is preferable. In either case, as has been discussed above, choosing a batch size below the points-to-pieces ratio of a dataset can be detrimental to performance, so this should be done only if instantaneous updates are a must.

Chapter 6

Related Work

Existing database systems provide some support for fitting models, but do not support regression models as first-class objects. In commercial DBMSs, this support typically takes the form of modeling tools and add-ons for data mining applications. For example, IBM's Intelligent Miner [14] and Oracle Data Miner [19] support creating models using PMML (Predictive Model Markup Language). However, these tools do not export a relational interface to model data. Rather, models are viewed as standalone black boxes with specialized interfaces for fitting and visualization. A typical use of PMML involving regression is to first fit a set of points to functions using an external tool, load those functions into the database, and then use the functions to predict the value of some other set of points by plugging them into the functions, typically using a stored procedure. This is very different than the approach proposed by this thesis, where the functions themselves can be joined, aggregated, and queried in conjunction with raw data stored in relational tables.

The work in [26] generalizes query optimization to support predicates that involve data mining models; however, [26] is mainly focused on classification, as opposed to regression models, which are our primary focus.

MauveDB [7] proposed querying models (including regression models) using a relational framework. FunctionDB is based on a similar idea, but extends the state of the art by using an algebraic framework and representation, which enable substantially faster and more accurate query execution than the gridding approach used by MauveDB.

The ideas of representing an infinite relation in a DBMS, and query processing using algebraic computations are not new. Constraint query languages, proposed in the context of querying geometric regions and formalized by [20], represent and query infinite regions as systems of constraints. There have been prototype implementations of constraint database systems for solving spatial queries and interpolating spatial data [6, 21, 22, 24, 25].

FunctionDB differs from constraint databases in two main ways. First, our data model is simpler and specifically restricted to regression models. Hence, our query processor is very different from a generalized constraint solver, and explicitly leverages the fact that tables contain functions that support algebraic primitives. FunctionDB is consequently more extensible to new classes of models, whereas constraint databases have focused mainly on linear constraints to keep query processing tractable. Second, the focus of our work is on efficient query processing for regression models, while work on constraint query languages and databases has traditionally focused on the complexity of supporting large numbers of constraints (*e.g.*, for linear programming applications).

Some systems (such as Postgres with PostGIS [5] extensions) support polyline and other trajectory data as ADTs. Internally, these types have a similar structure to the curves output by regression, but they do not support the range of operations over functional data that our more general model supports and are targeted exclusively towards geo-spatial data. Hence, these GIS extensions cannot, for example, compute the average value of a curve over some range, join two curves together, or convert to or from curves and discrete points.

Moving objects databases [12, 18] have proposed query languages and representations suitable for expressing queries on continuously varying spatio-temporal data, including trajectories. While some of their techniques have parallels with our ideas for querying continuous data, their work is specifically targeted at representing and querying object trajectories, while our work aims to support a more general class of applications using regression. Specific algorithms for similarity and indexing using piecewise line segments and functions have also been investigated earlier for time series and spatial data *e.g.*, [13] and [27]. These also are interesting instances of a general class of applications supported by FunctionDB.

Tools like MATLAB [1] support fitting regression functions, as well as algebraic and symbolic manipulation of functions, but lack support for relational queries. Also, as argued in [7], using these tools is inconvenient if data is already in a DBMS, because data needs to be moved back and forth between the external tool and the DBMS, resulting in considerable inconvenience and performance overhead.

Algorithms for online updates to time series models have been studied widely, and in several application contexts (*e.g.*, [10, 11, 16, 23]). The update algorithm proposed in Chapter 5 of this thesis, `BSearch`, draws on and adapts this body of previous work to devise an update algorithm generalizing to a wide class of regression functions, with an aim to meet the dual design goals of update performance and model quality in the context of streaming updates. Our approach, while similar in spirit to top down strategies proposed in previous work, generates more compact models by leveraging search.

Chapter 7

Conclusion

This thesis described and evaluated FunctionDB, a novel DBMS that supports regression functions as a data types that can be queried and manipulated like traditional relations. The thesis proposed a simple piecewise function representation for regression models, and showed how to build an extensible algebraic query processor by expressing relational operations in terms of basic algebraic primitives on functions. We have evaluated and quantified the benefits of algebraic query processing on two real-world applications that use modeling, and have shown that regression provides significant accuracy benefits over querying raw data directly. In addition, we have shown that our query processor is 10x-100x faster, as well as up to 15% more accurate on several realistic queries, compared to existing approaches that represent models as gridded data. The thesis has also illustrated and evaluated the benefits of batch updates when maintaining regression models online, in terms of both model compactness and accuracy of fit.

Bibliography

[1] Matlab. `http://www.mathworks.com/products/matlab/`.

[2] Mathematica. `http://www.wolfram.com/products/mathematica/index.html`.

[3] GNU Octave. `http://www.gnu.org/software/octave/octave.html`.

[4] The R Project For Statistical Computing. `http://www.r-project.org/`.

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

[6] Alexander Brodsky, Victor E. Segal, Jia Chen, and Pavel A. Exarkhopoulo. The CCUBE Constraint Object-Oriented Database System. In *SIGMOD Conference on Management of Data*, 1999.

[7] Amol Deshpande and Samuel Madden. MauveDB: Supporting Model-Based User Views in Database Systems. In *ACM SIGMOD Conference on Management of Data*, 2006.

[8] Bret Hull, Vladimir Bychkovsky, Yang Zhang, Kevin Chen, Michel Goraczko, Allen K. Miu, Eugene Shih, Hari Balakrishnan, and Samuel Madden. CarTel: A Distributed Mobile Sensor Computing System. In *4th ACM SenSys*, Boulder, CO, November 2006.

[9] E. F. Codd. A Relational Model Of Data For Large Shared Data Banks. *Communications of the ACM*, 26(1):64–69, 1983.

[10] R. O. Duda and P. E. Hart. *Pattern Classification And Scene Analysis.* Wiley, New York, 1973.

[11] Eamonn J. Keogh, Selina Chu, David Hart, and Michael J. Pazzani. An Online Algorithm For Segmenting Time Series. In *ICDM*, pages 289–296, 2001.

[12] Ralf Hartmut Guting, Michael H. Bohlen, Martin Erwig, Christian S. Jensen, Nikos A. Lorentzos, Markus Schneider, and Michalis Vazirgiannis. A Foundation for Representing and Querying Moving Objects. *ACM Transactions on Database Systems*, 25(1):1–42, 2000.

[13] Huanmei Wu, Betty Salzberg, Gregory C Sharp, Steve B Jiang, Hiroki Shirato, and David Kaeli. Subsequence Matching on Structured Time Series Data. In *ACM SIGMOD Conference on Management of Data*, 2005.

[14] IBM. IBM DB2 Intelligent Miner. `http://www-306.ibm.com/software/data/iminer/`.

[15] Jacob Cohen, Patricia Cohen, Stephen G. West, and Leona S. Aiken. *Applied Multiple Regression/Correlation Analysis for the Behavioral Sciences.* 2002.

[16] Jim Hunter and Neil McIntosh. Knowledge-Based Event Detection in Complex Time Series Data. In *Proceedings of the Joint European Conference on Artificial Intelligence in Medicine and Medical Decision Making*, pages 271–280, 1999.

[17] John Fox. *Applied Regression Analysis, Linear Models, and Related Methods.* 1997.

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

[19] Oracle. Oracle Data Miner. `http://www.oracle.com/technology/products/bi/odm/odminer.html`.

[20] Paris C. Kanellakis, Gabriel M. Kuper, and Peter Z. Revesz. Constraint Query Languages. In *Symposium on Principles of Database Systems*, pages 299–313, 1990.

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

[22] Peter Z. Revesz, Rui Chen, Pradip Kanjamala, Yiming Li, Yuguo Liu, and Yonghui Wang. The MLPQ/GIS Constraint Database System. In *SIGMOD Conference on Management of Data*, 2000.

[23] U. Ramer. An Iterative Procedure For The Polygonal Approximation Of Plane Curves. In *Computer Graphics And Image Processing*, volume 1, pages 244–256.

[24] Stéphane Grumbach, Philippe Rigaux, and Luc Segoufin. The DEDALE system for complex spatial queries. In *SIGMOD Conference on Management of Data*, pages 213–224, 1998.

[25] Stephane Grumbach, Philippe Rigaux, and Luc Segoufin. Manipulating Interpolated Data is Easier than You Thought. In *The VLDB Journal*, pages 156–165, 2000.

[26] Surajit Chaudhuri, Vivek R. Narasayya, and Sunita Sarawagi. Efficient Evaluation of Queries with Mining Predicates. In *International Conference on Data Engineering*, 2002.

[27] Yuhan Cai and Raymond Ng. Indexing Spatio-Temporal Trajectories with Chebyshev Polynomials. In *ACM SIGMOD Conference on Management of Data*, 2004.