MauveDB: Supporting Model-based User Views in Database Systems

Amol Deshpande, University of Maryland

Samuel Madden, MIT

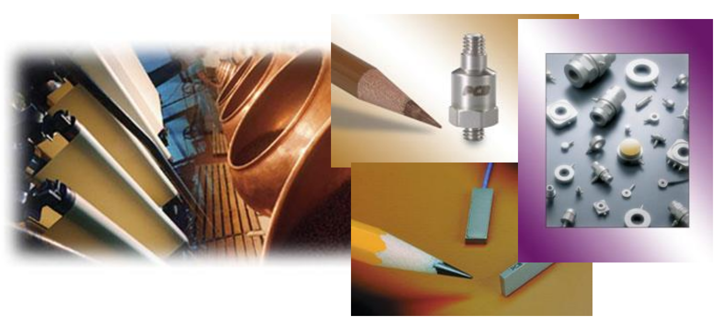
Motivation

- Unprecedented, and rapidly increasing, instrumentation of our every-day world

Wireless sensor networks

Distributed measurement networks (e.g. GPS)

RFID



Industrial Monitoring

Motivation

- Unprecedented, and rapidly increasing, instrumentation of our every-day world
  - Overwhelmingly large raw data volumes generated *continuously*
  - Data must be processed in *real-time*
  - The applications have strong *acquisitional* aspects
    - Data may have to be actively acquired from the environment
  - Typically *imprecise*, *unreliable* and *incomplete* data
    - Inherent measurement noises (e.g. GPS) and low success rates (e.g. RFID)
    - Communication link or sensor node failures (e.g. wireless sensor networks)
    - Spatial and temporal biases because of measurement constraints
- Traditional data management tools are ill-equipped to handle these challenges

Example: Wireless Sensor Networks

User

select *time, avg(temp)*
from *sensors*
epoch *1 hour*

{10am, 23.5}
{11am, 24}
{12pm, 30}

time	id	temp
10am	1	20
10am	2	21
..	..	…
10am	7	29

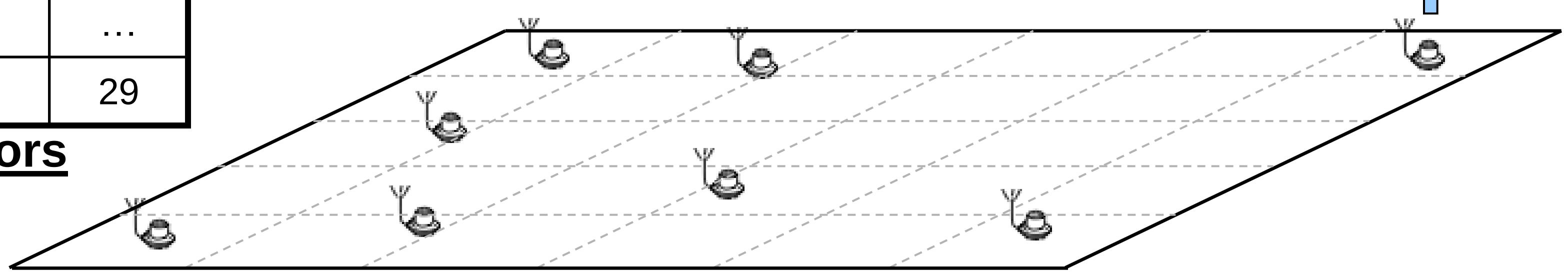
sensors

1. Spatially biased deployment
➔ these are not true averages

2. High data loss rates
➔ averages of different sets of sensors

3. Measurement errors propagated to the user

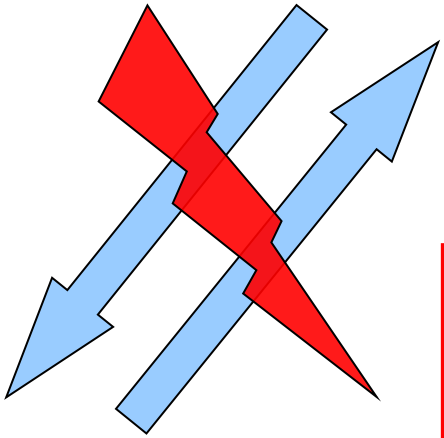
{12pm, 70}



A wireless sensor network deployed to monitor temperature

Example: Wireless Sensor Networks

User

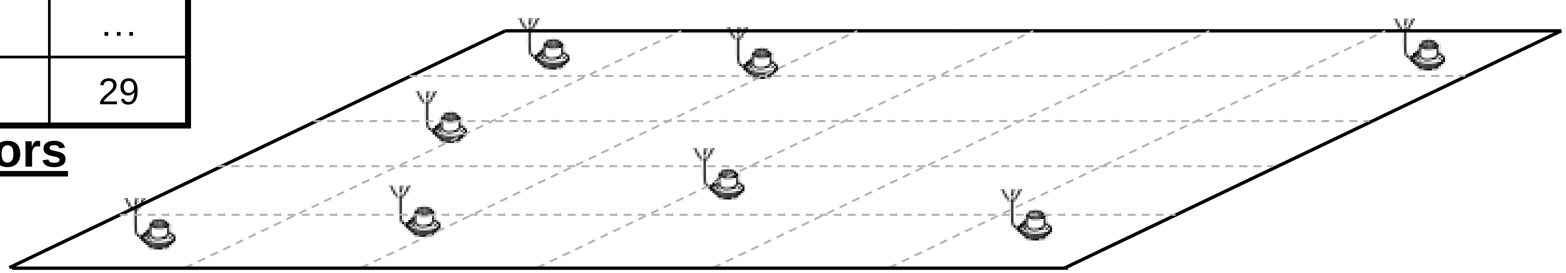


Impedance mismatch

User wants to query the "underlying environment", and not the sensor readings at selected locations

time	id	temp
10am	1	20
10am	2	21
..	..	…
10am	7	29

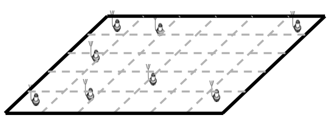
sensors



A wireless sensor network deployed to monitor temperature

Typical Solution

- Process data using a statistical/probabilistic model before operating on it
  - Regression and interpolation models
    - To eliminate spatial or temporal biases, handle missing data, prediction
  - Filtering techniques *(e.g. Kalman Filters)*, Bayesian Networks
    - To eliminate measurement noise, to infer hidden variables etc



insert into raw-data …

Table raw-data

<u>time</u>	<u>id</u>	<u>temp</u>
10am	1	20
10am	2	21
..	..	…
10am	7	29

*select * from raw-data*

raw-data tuples

1. Extract all readings into a file
2. Run a statistical model (e.g. regression) using MATLAB
3. Write output to a file
4. Write data processing tools to process/aggregate the output

<u>Sensor Network</u> *<u>Database</u>* *<u>User</u>*

Databases typically only used as a backing store; All data processing done outside

Issues

- Can't exploit commonalities, reuse/share computation
- No easy way to keep the model outputs up-to-date
- Lack of declarative languages for querying the processed data
- Large amount of duplication of effort
- Non-trivial
  - Expert knowledge & MATLAB familiarity required !

- Prevents real-time analysis of the data in most cases
- Why are databases not doing any of this ?
  - We are very good at most of these things

Solution: Model-based User Views

- An abstraction analogous to *traditional database views*
- Provides *independence from the messy measurement details*

No difference from a user's perspective

A *traditional* database *view* (defined using an SQL query)

User

avg-balances
select zipcode, avg(balance)
from accounts
group by zipcode

acct-no	balance	zipcode
101	a	20001
102	b	20002
..	..	
..	..	

accounts

A *model-based* database *view* (defined using a statistical model)

User

temperatures
Use *Regression* to predict missing values and to remove spatial bias

time	id	temp
10am	1	20
10am	2	21
..	..	…
10am	7	29

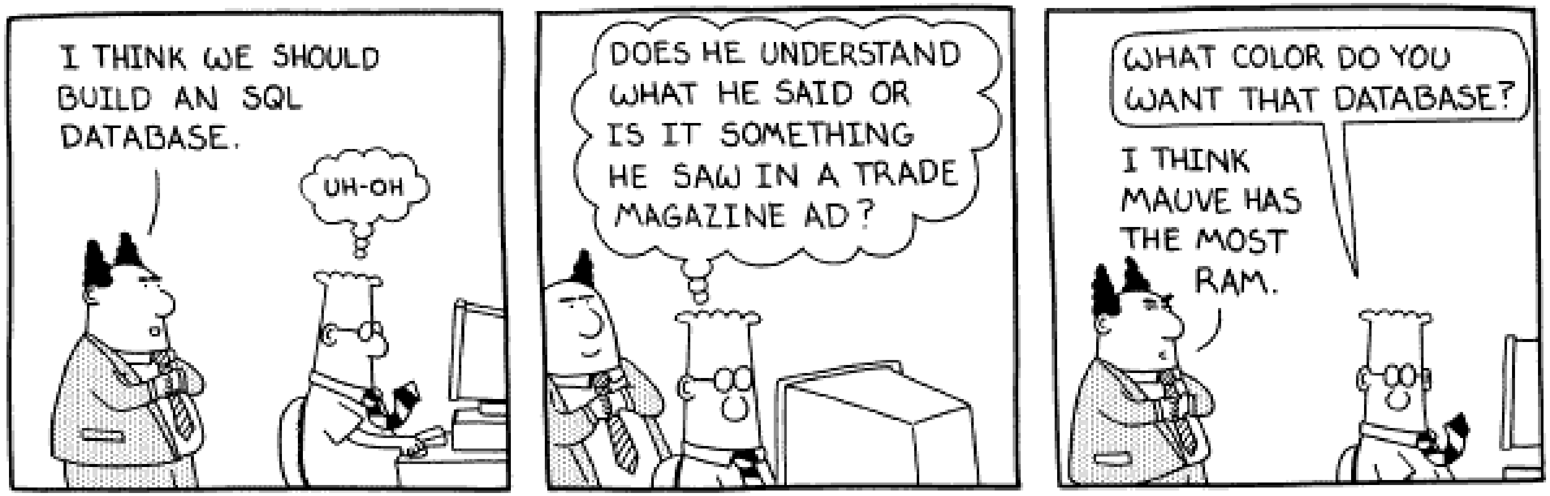
raw-temp-data

MauveDB System

- Supports the abstraction of Model-based User Views
- Provides declarative language constructs for creating such views
- Supports SQL queries over model-based views
- Keeps the models up-to-date as new data is inserted into the database

MauveDB System

- Supports the abstraction of Model-based User Views
- Provides declarative language constructs for creating such views
- Supports SQL queries over model-based views
- Keeps the models up-to-date as new data is inserted into the database



Outline

- Motivation
- Model-based views
  - Details, view creation syntax, querying
- Query execution strategies
- MauveDB implementation details
- Experimental evaluation

Linear Regression

- Models a ***dependent variable*** as a function of a set of ***independent variables***

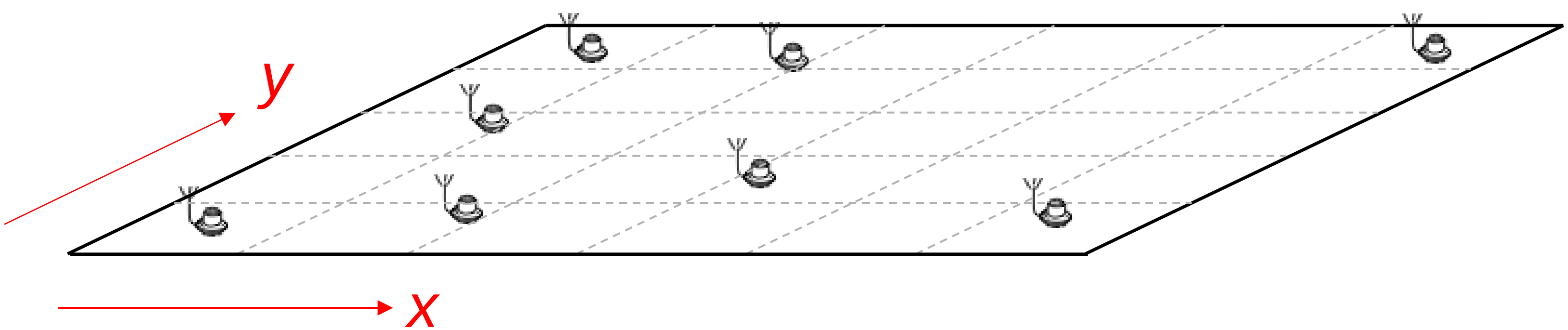
Basis Functions

Model *temperature* as a function of *(x, y)*

E.g.

$$temp = w_1 + w_2 * x + w_3 * x^2 + w_4 * y + w_5 * y^2$$

Weights



Grid Abstraction

A *Regression-based View*

User

Continuous Function

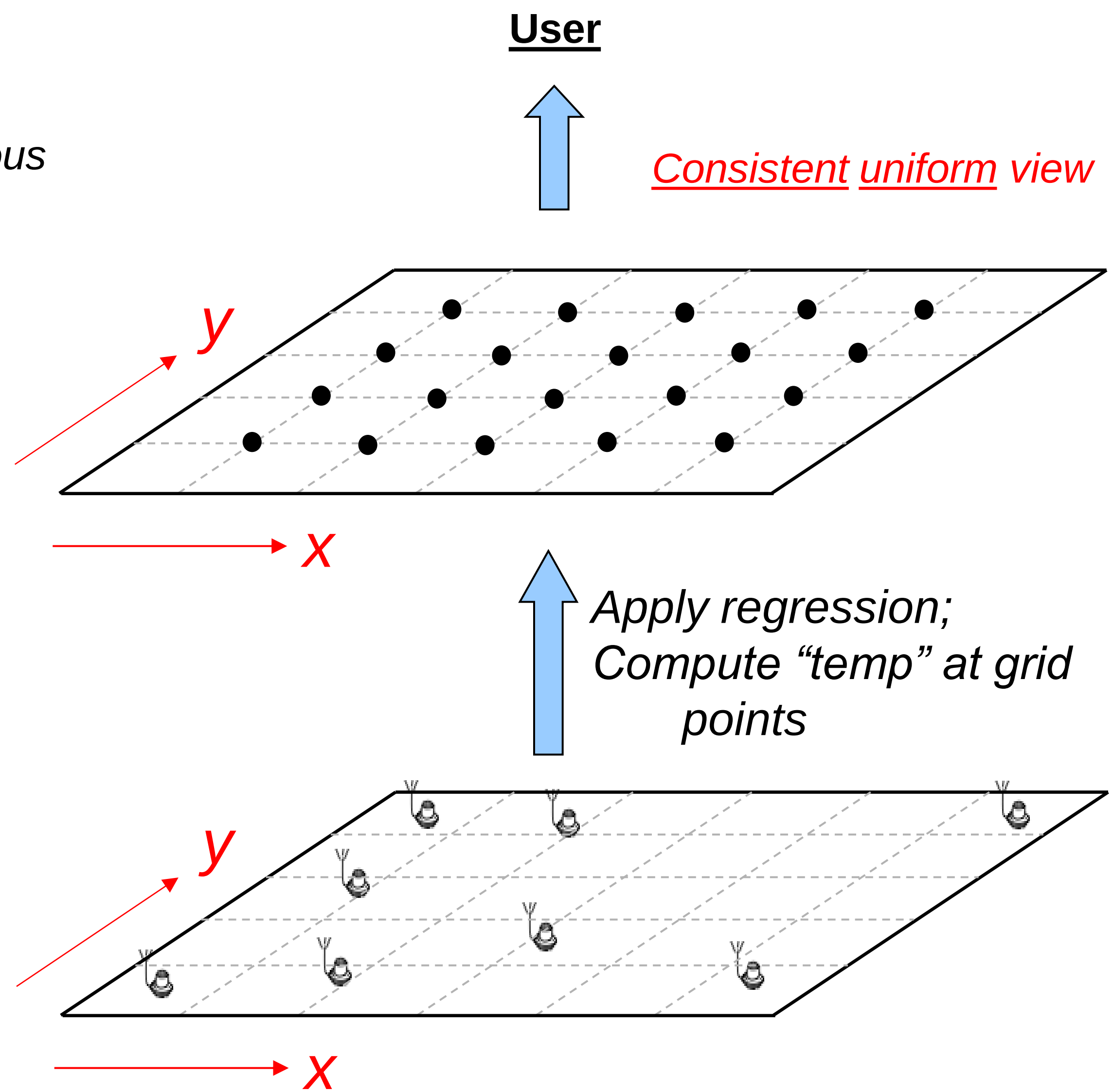
temperatures

Use *Regression* to model temperature as:

$$temp = w_1 + w_2\,x + w_3\,x^2 + w_4\,y + w_5\,y^2$$

time	id	temp
10am	1	20
10am	2	21
..	..	…
10am	7	29

raw-temp-data



Creating a Regression-based View

Matlab-like syntax used for specifying the grid

CREATE VIEW

RegView(time [0::1], x [0:100:10], y[0:100:10], temp)

Schema of the View

AS

FIT temp USING time, x, y

BASES 1, x, x^2, y, y^2

Model to be used

FOR EACH time T

TRAINING DATA

SELECT temp, time, x, y

FROM raw-temp-data

WHERE raw-temp-data.time = T

Training data for learning parameters

View Creation Syntax

- Somewhat model-specific, but many commonalities

<u>A Interpolation-based View</u>

CREATE VIEW

IntView(t [0::1], sensorid [::1], y[0:100:10], temp)

AS

INTERPOLATE temp USING time, sensorid

FOR EACH sensorid M

TRAINING DATA

SELECT temp, time, sensorid

FROM raw-temp-readings

WHERE raw-temp-readings.sensorid = M

Outline

- Motivation
- Model-based views
  - Details, view creation syntax, querying
- Query execution strategies
- MauveDB implementation details
- Experimental evaluation

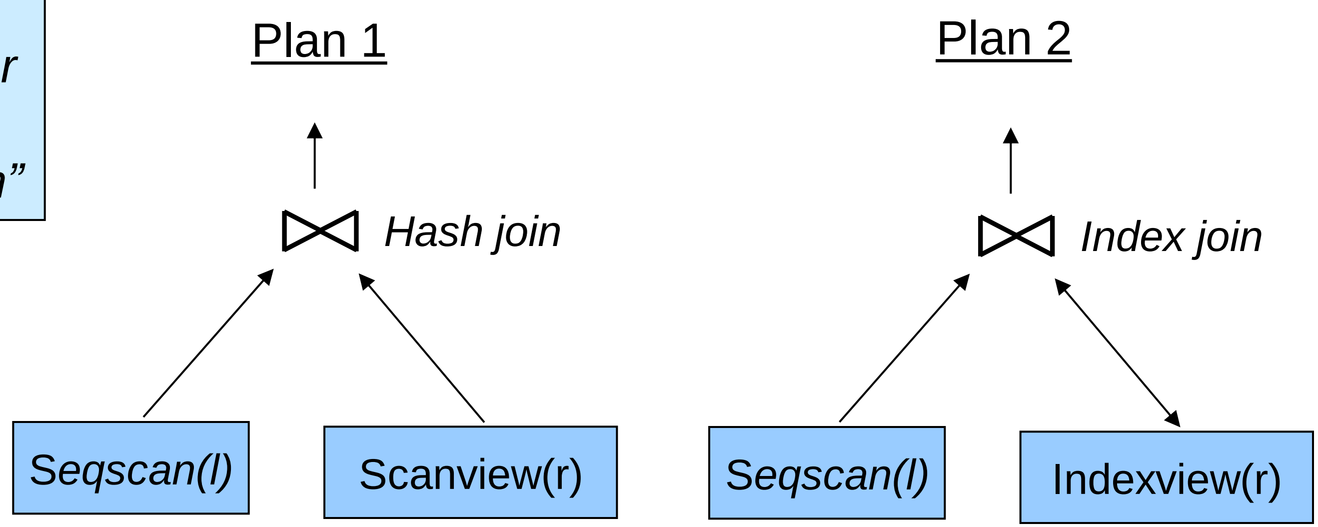
Querying a Model-based View

- Analogous to traditional views
- So:
  - *select * from reg-view*
    - Lists out temperatures at all grid-points
  - *select * from reg-view where x = 15 and y = 20*
    - Lists temperature at (15, 20) at all times
  - …

Query Processing

- Two operators per view type that support *get_next() API*
  - *ScanView*
    - Returns the contents of the view one-by-one
  - *IndexView (condition)*
    - Returns tuples that match a condition
      - e.g. return *temperature* where *(x, y) = (10, 20)*

*select **
from locations l, reg-view r
where (l.x, l.y) = (r.x, r.y)
and r.time = "10am"



View Maintenance Strategies

- Option 1: Compute the view as needed from base data
  - For regression view, scan the tuples and compute the weights
- Option 2: Keep the view materialized
  - Sometimes too large to be practical
    - E.g. if the grid is very fine
  - May need to be recomputed with every new tuple insertion
    - E.g. a regression view that fits a single function to the entire data
- Option 3: Lazy materialization/caching
  - Materialize query results as computed

- Generic options shared between all view types

View Maintenance Strategies

- Option 4: Maintain an efficient *intermediate representation*
- Typically model-specific

- Regression-based Views
  - Say $temp = f(x, y) = w_1\, h_1(x, y) + \ldots + w_k\, h_k(x, y)$
  - Maintain the *weights* for *f(x, y)* and a *sufficient statistic*
    - Two matrices ($O(k^2)$ *space)* that can be incrementally updated
  - ScanView: Execute *f(x, y)* on all grid points
  - IndexView: Execute *f(x, y)* on the specified point
  - InsertTuple: Recompute the coefficients
    - Can be done very efficiently using the sufficient statistic

- Interpolation-based Views
  - Build and maintain a tree over the tuples in the *TRAINING DATA*

Outline

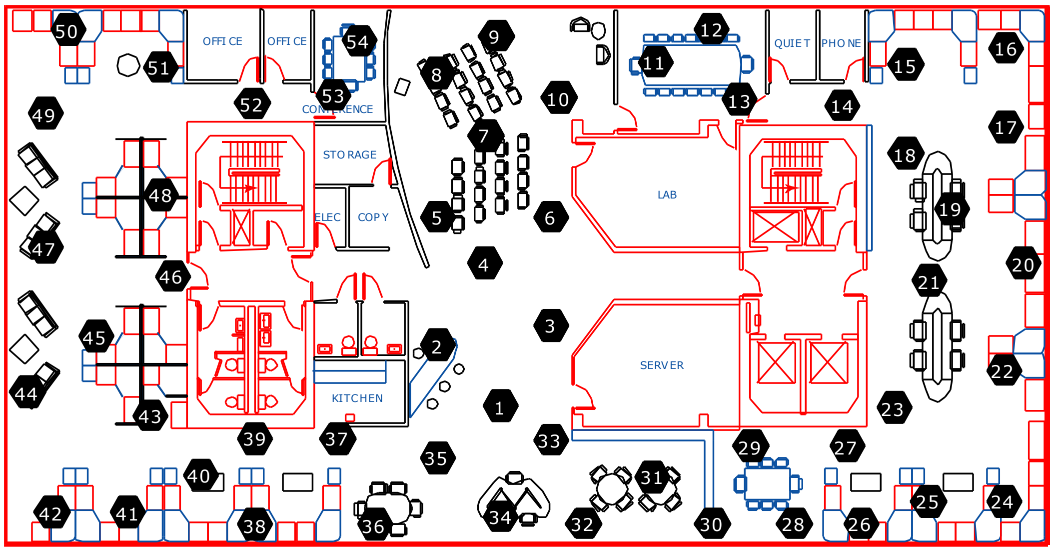
- Motivation
- Model-based views
  - Details, view creation syntax, querying
- Query execution strategies
- MauveDB implementation details
- Experimental evaluation

MauveDB: Implementation Details

- Written in the Apache Derby Java open source database system
- Support for *Regression-* and *Interpolation-based views*
- Minimal changes to the main codebase
- Much of the additional code (approx 3500 lines) fairly generic in nature
  - A view manager (for bookkeeping)
  - Query processing operators
  - View maintenance strategies
- Model-specific code
  - Intermediate representation
  - Part of the view creation syntax

MauveDB: Experimental Evaluation

- Intel Lab Dataset
  - 54-node sensor network monitoring *temperature, humidity etc*
  - Approx 400,000 readings
  - Attributes used
    - Independent - *time, sensorid, x-coordinate, y-coordinate*
    - Dependent - *temperature*



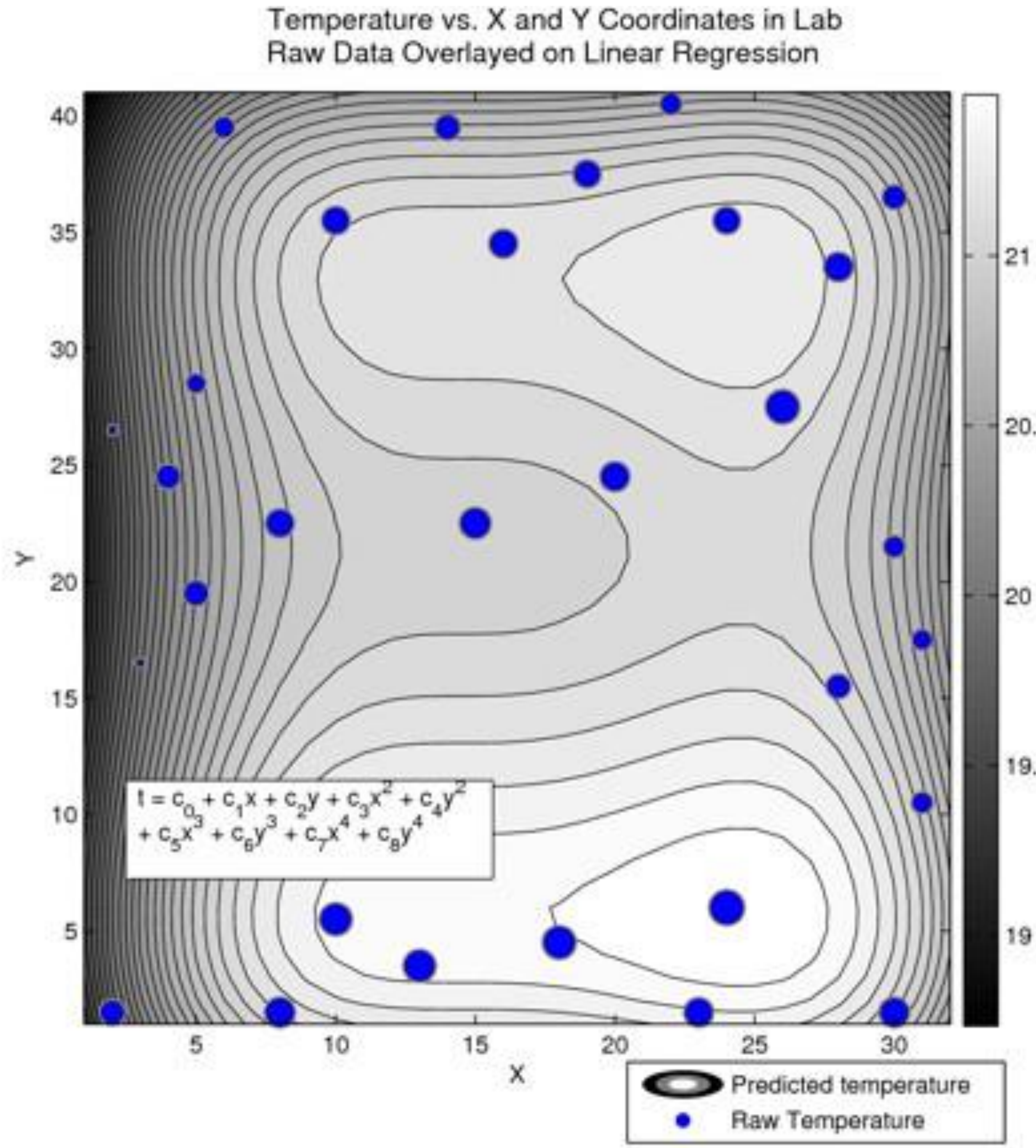
Spatial Regression

Contour plot over the data obtained using:

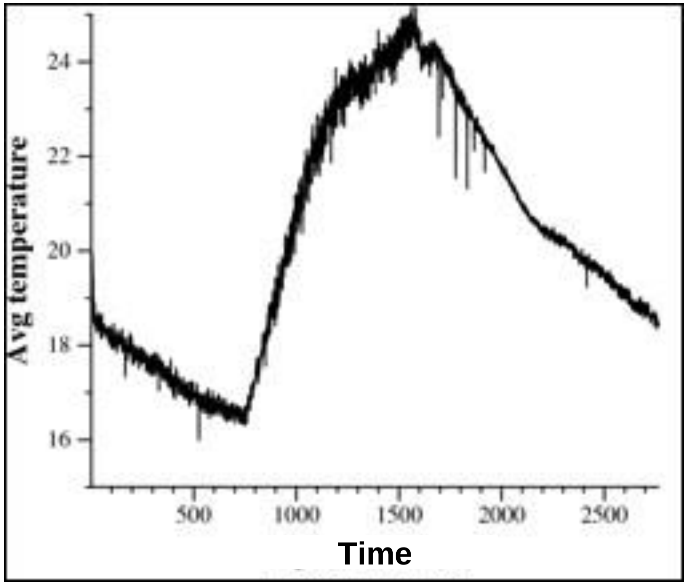
*select **

from reg-view

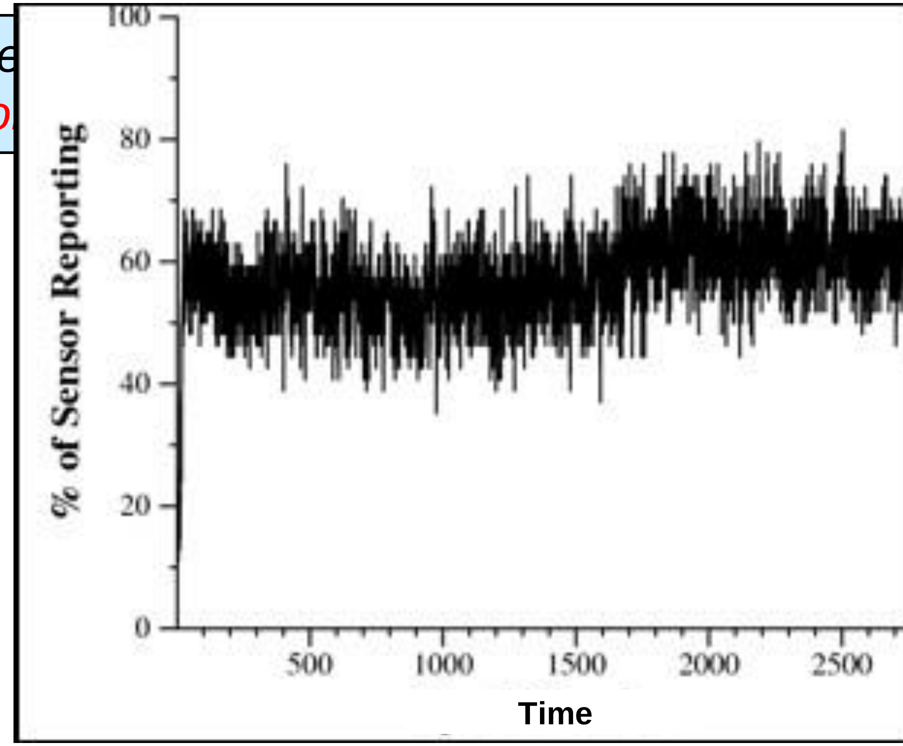
where time = 2100



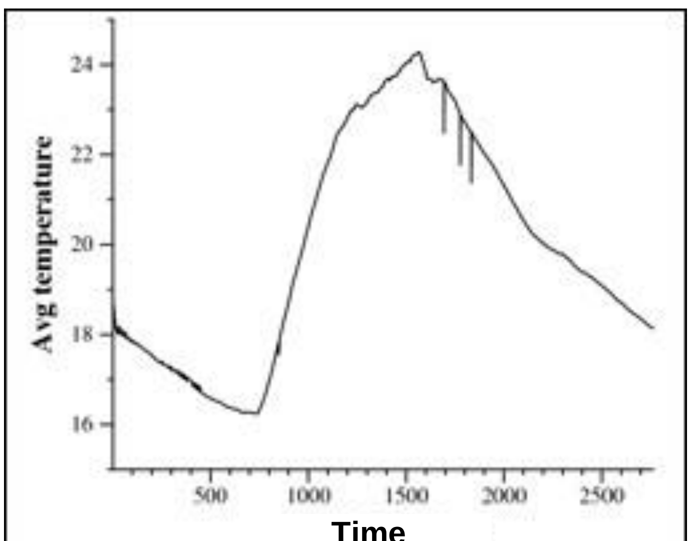
Interpolation



Average te
raw senso



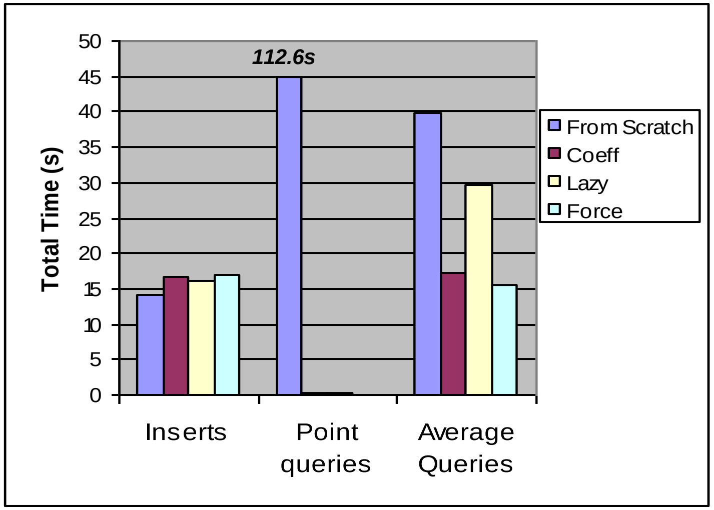
Over 40% missing data



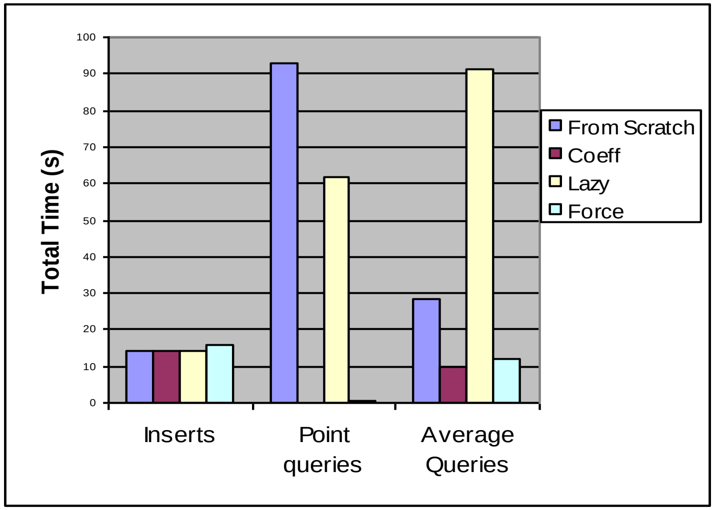
Average temperature over an interpolation-view over the raw sensor readings

Comparing View Maintenance Options

- 50000 tuples initially
- Mixed workload:
  - insert 1000 records
  - issue 50 point queries
  - issue 10 average queries
- Brief summary:
  - Intermediate representation typically the best
  - Among others, dependent on the view properties, and query workload



Regression, per time



Interpolation, per sensor

Ongoing and Future Work

- Adding support for views based on *dynamic Bayesian networks (e.g. Kalman Filters)*
  - A very general class of models with wide applicability
  - Generate *probabilistic data*
- Developing APIs for adding arbitrary models
  - Minimize the work of the model developer
- Query processing, query optimization, and view maintenance issues

- Much research still needs to be done

Conclusions

- Proposed the abstraction of model-based views
  - Poweful abstraction that enables declarative querying over noisy, imprecise data
- Exploit commonalities *to define, to create, and to process queries* over such views
- MauveDB prototype implementation
  - Using the Apache Derby open source DBMS
  - Supports *Regression- and Interpolation-based views*
  - Supports many different view maintenance strategies

Thank you !!

- Questions ?