Contents

Chapter 1

Introduction

Geographic Information Systems (GISs) are specialized computer systems for the storage, retrieval, manipulation, analysis and display of large volumes of spatial or map type data [36].

Many GIS systems contain spatiotemporal data such that the geographically distributed values change continuously over time. These continuously changing data will create infinite number of tuples in traditional relational databases. Constraint databases [27] provide a natural way to represent them such that these infinite values can be represented in finite constraint tuples.

There are now a number of implemented constraint database systems, such as CCUBE [3], DEDALE [10] and MLPQ [28]. The architecture of these constraint database systems are very similar. They all contain several special features which never occur in relational and object-oriented database system design. These features may include modules for constraint representation, data approximation and special data visualization.

Value-by-area cartograms provide a highly expressive visualization of geographically distributed data. For continuously changing data, an animation can be done by successive displaying of value-by-area cartograms at different time instances. Such an animation can reveal more information than is revealed by only a few selected value-by-area snapshots. In this dissertation we describe several value-by-area cartogram animation algorithms that can be used to visualize geographically distributed continuous spatiotemporal data that often occur in GIS systems. We implemented the algorithms as part of the graphical user interface of the MLPQ/GIS database system.

In GIS systems (and also many other systems), continuously changing data are often measured and recorded only sporadically as time series data. Since a fine granularity of time may be needed, the traditional representation of time series data, as a set of data points, requires too much computer storage space and allows only inefficient data retrieval and querying. We propose more efficient alternative representations for time series data: piecewise linear approximation. In the piecewise linear approximation, the time series data s is approximated by a piecewise linear function f, such that at each time instance, the error between actual data and the approximate data is less than some fixed error tolerance.

Piecewise linear approximation is very important in animation because it provides interpolation of data, enabling the animation to be done in any time granularity. The data compression in piecewise linear approximation also helps the evaluation process to work faster.

Query optimization is very important in query evaluation [18, 24, 29, 32, 35]. Wong and Youssefi's query optimization algorithm [35] for relational algebra queries is a well-known optimization algorithm that forms the basis of several systems. In this dissertation we propose a new query optimization algorithm which is an improvement of Wong and Youssefi's algorithm.

1.1 Outline of the Dissertation

The rest of this dissertation is organized as follows. Chapter 2 gives a brief introduction on relational and constraint databases, as well as reviewing the previous work on cartogram animations and query optimizations.

Chapter 3 describes several methods for value-by-area cartogram animation and a new algorithm for creating single value-by-area cartogram. We present the algorithm and methods such that they can be used to construct a cartogram animation system that works fast enough to avoid pre-computing and saving of the snapshots. This enables us to animate databases that have a large number of snapshots. This chapter is based on our work of [20, 21].

Chapter 4 discusses the piecewise linear approximation for GIS time series data. We propose an $O(n)$ complexity algorithm that can compress the original time series data while guaranteeing certain maximal error tolerance. It is based on the work of [4, 5, 26].

Chapter 5 tries to combine Chapter 3 and Chapter 4 to explore the usage of piecewise linear approximation in the value-by-area cartogram animation. The interpolation and data compression abilities in piecewise linear approximation help to make more smooth and more efficient value-by-area cartogram animation.

Chapter 6 is based on [6]. It discusses the approximate evaluation of queries based on the piecewise linear approximation. We show that approximate evaluation can get high precision and recall while creating much less constraint tuples.

Chapter 7, based on [19, 22], proposes a new hypergraph partitioning based query optimization strategy whose evaluations require less space for the intermediate size relations than the optimized expressions given by earlier algorithms. It also gives parallel evaluation strategies which can be more efficient in parallel computers.

Finally, Chapter 8 concludes the dissertation with some directions for future work.

Chapter 2

Background

2.1 Relational and Constraint Database Systems

Database management system is a very important component of a modern computer system. In its early ages in the 1960s, the network and hierarchical models were widely used. Now, the relational database model has become the primary model for commercial databases. For relational database model, there are some theories that assist the design of databases and the efficient processing of queries.

The relational database model was first proposed by Codd in 1970. In relational database model, a database is a collection of one or more *relations*. Each *relation* has a unique name and can be represented by a table with rows and columns [29].

In relational databases, each relation consists of a **relation schema** and a relation instance. The **relation schema** describes the design of the relation (table), such as the relation name, the name and domain of each field of the relation (table). A *relation instance* is a table as the instance of the *relation schema.* The following Table 2.1 gives a relation instance for the relation schema

$$CSE413\_Scores = (name, SSN, score)$$

which record the scores of each student in the course of "CSE413".

Example 2.1.1 Relational Database table for relation $CSE413\_Scores = (name,$ $SSN,\ score)$:

name	SSN	score
Amy	999-99-0001	90
Billy	999-99-0017	85
Charlie	999-99-0117	88
Dick	999-99-0227	92
John	999-99-3227	98

Table 2.1: Database for Relation $CSE413\_Scores$

Besides relational database representations, an alternative way to represent the data is to use constraint databases. The constraint data model provides a finite representation of the unrestricted relational data model. The essential difference is that in the constraint data model, the value of any attribute is specified implicitly using variables and constraints [27].

Example 2.1.2 Suppose in a single day we record the precipitation in a city with area $4 * 10^{12}$ (square inches). It began to rain at 9:00am, 0.5 inch per hour and the rain stopped at 1:00pm. At 7:30pm, it began to rain 0.75 inch per hour and stopped at 11:30pm. Then the amount of rainfall in this city (in cubic inches) is shown in Figure 2.1

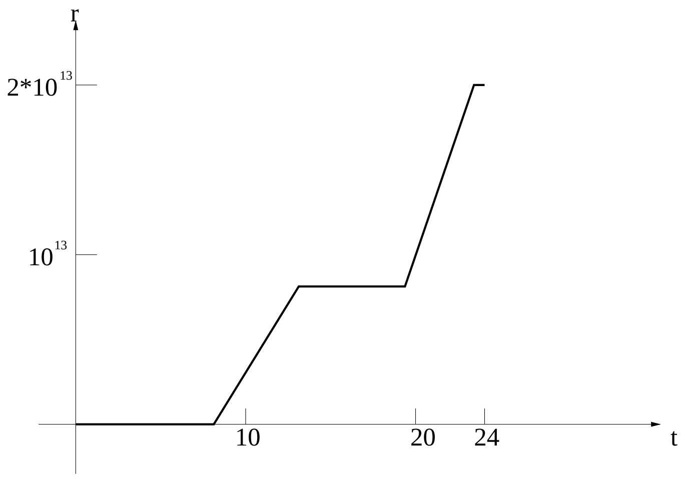


Figure 2.1: Amount of Rainfall

Suppose that the amount of rainfall at each time instance is represented in the following relation schema:

$$Amount\_of\_Rainfall = (time, rainfall)$$

In the unrestricted relational data model, a database instance could contain an infinite number of tuples. The rainfall relation contains an infinite number of tuples because the time instance can be any real number between 0 and 24, as shown in Figure 2.2.

The constraint relation for the above example is shown in Table 2.3. This

time	rainfall
00:00	0.0
...	
10:00	$2.0 * 10^{12}$
...	
21:30	$1.4 * 10^{13}$
...	
24:00	$2.0 * 10^{13}$

Table 2.2: Unrestricted Relational Database for Rain

constraint relation describes the same relation as in Table 2.2. However, the representation is different. In constraint relation, it uses a variable t as the value of the time and the valuable r as the rainfall. Both variables range over the real numbers. The first row of the of relation means that the rainfall is r at the time t, if the condition $0 \leq t \leq 9.0, r = 0.0$ is true. We call such a condition a *constraint*. Each constraint can be a list of atomic constraints which are separated by commas that are read "and". The first constraint expresses that up to time 9.0 (9:00am), the amount of rainfall is 0. The next row expresses the rainfall is $2.0 * 10^{12} * (t - 9)$ when t is between 9.0 (9:00am) and 13.0 (1:00pm). Other rows can be explained similarly.

Note that the entire constraint relation consisted of only five constraint tuples although it represents the infinite relation described earlier.

Having some databases in a computer system, a user can use query languages to request information from them. Relational algebra is one option, it consists of a set

time	rainfall	
t	r	$0 \le t \le 9.0, r = 0.0$
t	r	$9.0 < t \le 13.0, r = 2.0 * 10^{12} * (t - 9)$
t	r	$13.0 < t \le 19.5, r = 8.0 * 10^{12}$
t	r	$19.5 < t \le 23.5, r = 8.0 * 10^{12} + 3.0 * 10^{12} * (t - 19.5)$
t	r	$23.5 < t < 24.0, r = 2.0 * 10^{13}$

Table 2.3: Constraint Database for Rainfall

of operations that take one or two relations as input and produce a new relation as their result. The fundamental operations in the relational algebra are *select, project, union, set difference, Cartesian product* and *rename.* other operations include *set intersection, natural join, division* and *assignment* [29].

In relational algebra, we use lowercase Greek letter sigma (σ) to denote selection, the uppercase Greek letter pi (Π) to denote project, $\bigcup$ for union, $-$ for set difference, $\times$ for Cartesian product and ρ for rename. We also use $\bigcap$ for intersection, $\bowtie$ for natural join and / for division. In this way, the query of finding the students whose scores are greater than 90 can be written as:

$$\Pi_{name}\ \sigma_{score>90}\ CSE413\_Scores \tag{2.1}$$

Relational algebra is a formal query language. However, commercial database systems usually use SQL language. SQL is not only a “query language”, it contains many capabilities besides querying a database. Including features for defining the structure of the database, modifying the database and for specifying security constraints [29].

The following is a SQL query that is equivalent to relational algebra in 2.1.

$$\begin{array}{ll} \textbf{Select} & name \\ \textbf{from} & CSE413\_Scores \\ \textbf{where} & score > 90 \end{array} \tag{2.2}$$

Another query language is Datalog, which is a rule-based language related to Prolog. In Datalog each rule is a statement saying that if some points belong to some relations, then other points must also belong to a defined relation. Each Datalog query contains a Datalog program and an input [28]. The Datalog query for the previous database retrieval problem can be written as:

$$High\_Scores(name) : -CSE413\_Scores(name, SSN, score),$$

$$score > 90.$$

Both the constraint databases and the relational databases can be queried by either relational algebra, SQL or Datalog.

2.2 Cartograms and Cartogram Animation

Many GIS databases contain spatiotemporal data such that the geographically distributed value change continuously over time. Population and precipitation distributions are two such examples. Value-by-area cartograms provide a highly expressive visualization for this kind of data. A value-by-area cartogram is created by dividing the original map into small areas (cells) and then enlarging or shrinking each cell to make its area proportional to its given "value". For example, a value-by-area cartogram where the cells are the continental U.S. states and the values are their populations in 1990 is shown in Figure 2.2, the population data comes from the US Census Bureau http://www.census.gov. In this value-by-area cartogram, each state's area is proportional to its population. By looking at the map, we can easily see the population distribution in the continental United States.

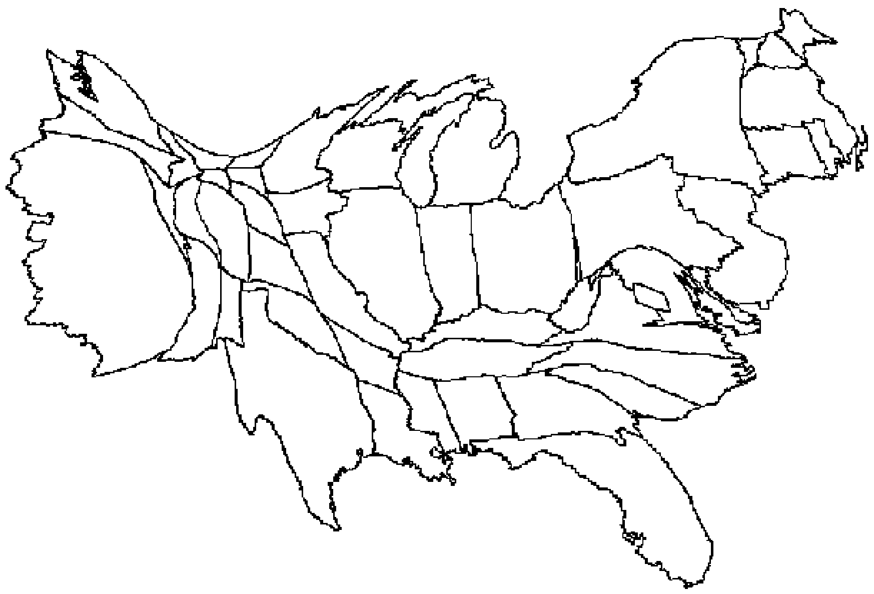


Figure 2.2: A Value-by-Area Cartogram for the U.S. Population in 1990

For the same set of data, there are several possible value-by-area cartograms.

In general, the easier it is to recognize which cells are which in a value-by-area cartogram the better (and more informative) it is. For example, for anyone familiar with the usual geographical map of the continental U.S. states, it is easy to recognize the individual states. In that way, one can learn that for example California has a high population compared to its area as it is enlarged, which Nebraska has a small population compared to its area as it is shrunk compared to the usual geographic map. It is in this way that cartograms can be informative and useful for people who do not wish to look at statistical tables.

There are two basic forms of cartograms: contiguous cartograms and noncontiguous cartograms. In contiguous cartograms, the internal cells are adjacent to each other, while in noncontiguous cartograms, this is not necessary the case. Figure 2.3 illustrates the original map with four cells, each has the geographic distributed value with it, the contiguous cartogram and the noncontiguous cartogram whose cell areas are proportional to the its "value".

Generally, contiguous cartograms looks better than noncontiguous cartograms because they look more like real maps, the geographic arrangement of cells also helps people to recognize the cells with respect to original geographic map. On the other hand, it is easier for noncontiguous map to preserve original shape of each cell, and they are easier to construct. Because if we do not care the continuity of the cartogram, we can simply enlarge/shrink each cell to its desired area.

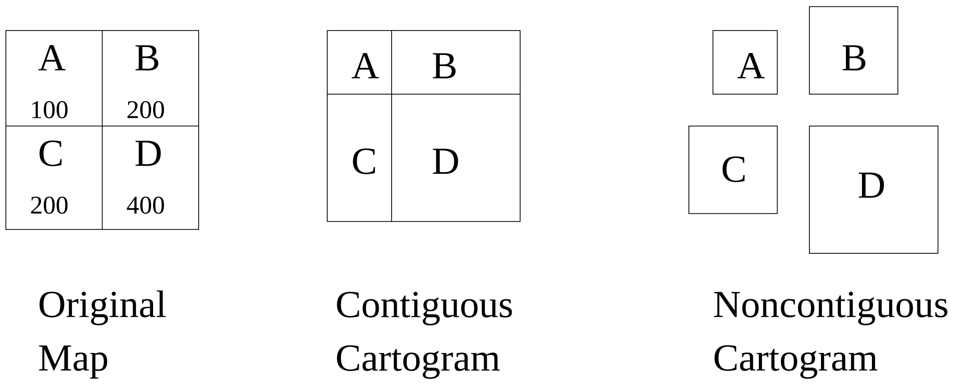


Figure 2.3: Contiguous and Noncontiguous Cartograms

Contiguous cartograms are much more difficult to construct. Usually the cells are represented by polygons. When trying to construct contiguous cartograms, each cell has to be enlarged/shrunken. The enlarge/shrink requirement for each cell may give conflict move direction for cell corner vertices. The cell shape has to be somehow distorted to keep the cartogram contiguity.

Cartograms can be drawn manually. However, computer programs are more desirable. From 1973, a few computer algorithms for creating contiguous value-by-area cartograms have been proposed [7, 13, 11, 30]. Among them, the pseudo-cartogram algorithm proposed by Tobler [30] is the earliest. Generally, pseudo-cartogram algorithm converges slowly (if it converges at all), and the area error is large.

Dougenik et al.'s algorithm [7] and Gusein-Zade et al.'s algorithm [11] are based on rubber-sheet transformation idea, which can give highly accurate cartograms, but the computation time is long.

House and Kocomoud [13] proposed another value-by-area algorithm. They view the construction of value-by-area cartogram as a constraint optimization problem, and they apply the simulated annealing approach to solve it. This algorithm runs quiet slow, according to [13], it will take hours to compute a 1980 United States population cartogram on a 300MHz Compaq Professional Workstation with 128MB memory.

For continuously changing data such as population or precipitation data the value-by-area cartogram animation successively displays value-by-area cartograms at different time instances. Such animations can reveal more information than is revealed by only a few selected value-by-area cartogram snapshots. For example, we did an animation for the monthly precipitation in the continental U.S. states between 1948 and 1998. The animation revealed that the precipitation has more regular cycles in the New England states than in the western states. Such animations allow everyone to make similar observations without even knowing anything about statistics.

Cartogram animation is a relatively new topic. Kocomoud and House [16] give an animation for the U.S. population cartograms from 1900 to 1996, which contains eleven cartogram snapshots of 1900, 1910, $\cdots$, 1990 and 1996. White et. al. [34] give an animation for the infant mortality in the United Kingdom from 1856 to

1925, which contains less than 10 snapshots. In both animations, the cartogram snapshots are pre-computed and saved for display. This pre-computation strategy is practical if an animation contains only a small number of snapshots. However, if an animation requires a large number of snapshots, it is not practical to pre-compute and save the snapshots. For example, if we want to study the monthly precipitation pattern from decades of precipitation data, then the animation may contain hundreds of snapshots, it will require too much space if we pre-compute and save the snapshots.

2.3 Query Optimization

Query optimization is a very important problem in database systems. Wong and Youssefi's query optimization algorithm [35] is a well-known optimization algorithm that forms the basis of several systems. It is based on the idea of hypergraph representation of the queries and on a particular reduction of the hypergraphs. In the reduction the edges are taken away from the hypergraph according to some rules and the order implies a particular sequence of operations.

Hypergraph Representation of Queries A hypergraph $H(V, E)$ is an extension of a graph in which $V = \{v_1, v_2, \cdots, v_n\}$ is the set of vertices, $E = \{e_1, e_2, \cdots, e_m\}$ is the set of hyperedges. Each hyperedge is a connection of two or more vertices. A graph can be seen as a special hypergraph such that each hyperedge connects exactly two vertices.

A query can be represented as a hypergraph [32]. The following is an example for a query on relations and the corresponding hypergraph.

Example 2.3.1 In the following GIS database, there are four relations:

WeatherStation(SN, longitude, latitude, state, county)

WeatherData(SN, date, highTemperature, lowTemperature, precipitation)

EnergyConsumption(state, county, date, electricConsumption)

Capital(state, capital)

In the above relations, $WeatherStation$ (SN, $longitude$, $latitude$, $state$, $county$) give the information of a weather station, which includes the station number (SN), the longitude and the latitude, the state and county in which the station locates. The relation $WeatherData$(SN, $date$, $highTemperature$, $lowTemperature$, $precipitation$) gives the daily weather data (highTemperature, lowTemperature, precipitation) for each day. The relation $EnergyConsumption$($state$, $county$, $date$, $electricConsumption$) gives the electric consumption data for each day and the relation $Capital$($state, capital$) gives the capital city for each state. Suppose that there is a weather station in each county and we would like to study the relations between the weather and the electric consumption in the capital city of Nebraska, we can use the following relation algebra query to list the weather and electronic consumption data data:

$$\Pi_{WeatherData.date,highTemperature,lowTemperature,precipitation,electricConsumption}$$

$$\sigma_{\substack{WeatherStation.SN=WeatherData.SN \ \wedge \ WeatherData.date=EnergyConsumption.date \ \wedge \\ WeatherData.state=EnergyConsumption.state \ \wedge \ WeatherData.county=EnergyConsumption.county \ \wedge \\ WeatherData.state=Capital.state \ \wedge \ WeatherData.county=Capital.capital \ \wedge \ Capital.state=``Nebraska''}}$$

$$(WeatherStation \times WeatherData \times EnergyConsumption \times Capital)$$

The above query can be represented by a hypergraph. In the hypergraph representation of a query, each attribute in the relation is represented by a single vertex, except if two attributes a and b have the equality condition $a = b$ in the query expression, then they are merged into one vertex. Each relation or select condition is represented by a hyperedge that contains the related attributes.

Wong-Youssefi's Optimization Algorithm: Wong and Youssefi [35] proposed a hypergraph based query optimization algorithm in which the query optimization is done by taking away from the hypergraph the hyperedges one at a time.

The algorithm tries to find "small hyperedges" that when taken away will disconnect the hypergraph. A hyperedge is considered small if it represents selections

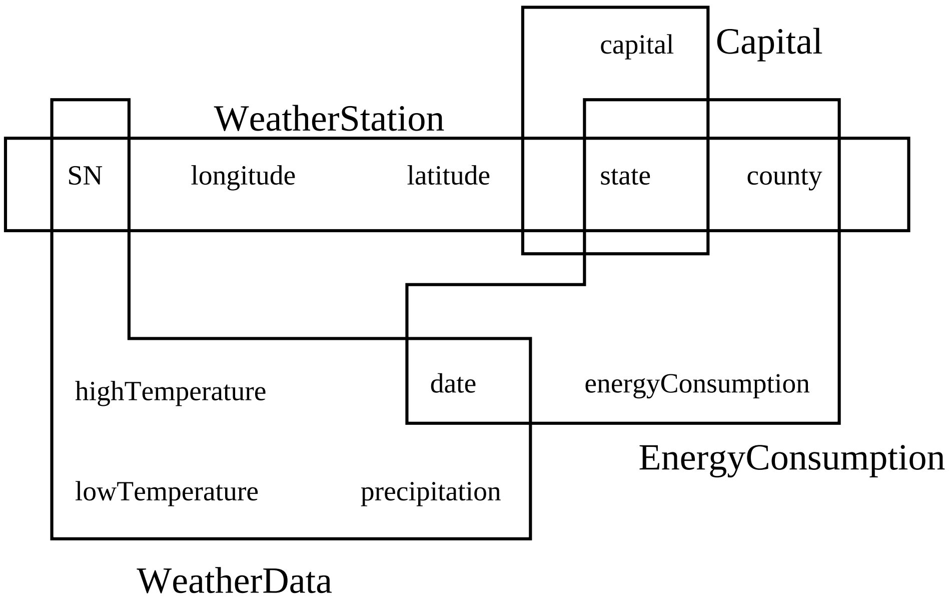


Figure 2.4: Hypergraph for SQL Query

with equality conditions between attributes and constants or were reduced by a semi-join operation with another relation. The algorithm gives preference to small hyperedges even if they do not disconnect the graph. If there is no disconnecting hyperedge, then just an arbitrary hyperedge is taken away.

In addition, whenever a hyperedge/relation is taken away, the Wong-Youssefi algorithm also takes a semi-join of it with other hyperedges/relations with which it shares some common attributes.

Chapter 3

Data Visualization

In this chapter, we propose several methods for value-by-area cartogram animation. Each animation method is based on a new algorithm for creating single value-by-area cartograms. The presentation below is based on our work in [20, 21].

We also implemented the animation system and integrated it into the MLPQ constraint database system[14] as one of the visual interface. It can provide highly expressive animations for real spatiotemporal databases.

The rest of this section is structured as follows. Section 3.1 discusses three different methods for creating sequences of value-by-area cartograms as well as a new fast value-by-area cartogram algorithm. Section 3.2 gives implementation results.

3.1 Animation with Large Number of Snapshots

Some cartogram animation models [16, 34] pre-compute and save the snapshots for display. Each of these models contains only a few cartogram snapshots, thus they can be pre-computed and saved for display. However, if the animation contains a large number of snapshots, then the pre-computation strategy is not practical. A better way is to compute the snapshots during animation. This strategy requires fast algorithms for snapshot computation.

3.1.1 Value-by-Area Cartogram Animation Methods

We describe three basic methods for value-by-area cartogram animation, which provides a visualization of geographically distributed continuous spatiotemporal data.

Each animation method uses as a basic procedure some value-by-area cartogram snapshot algorithm. Each snapshot algorithm takes as input a map with some cell division and values for each cell and gives as output a distorted map in which each cell has the same cell density (the "density" is the result of cell value divided by cell area). Independent of the basic snapshot procedure used, the three animation methods work in different ways.

Parallel Method: It is possible to apply the algorithms described in [7, 11, 13, 30, 31] to compute the cartogram snapshots for animation. This is the idea behind the *Parallel Method*, which is one way to compute the series of cartogram snapshots. The *Parallel Method* constructs each cartogram snapshot based on the original geographic map. We call it *Parallel Method* since each snapshot can be computed independently of the other snapshots and thus they can be computed in parallel. In theory, the *Parallel Method* can be used together with any of the algorithms described in [7, 11, 13, 30, 31]. But actually these algorithms are too slow for animation. We describe a faster algorithm in Section 3.1.2. An illustration of the *Parallel Method* is shown in Figure 3.1.

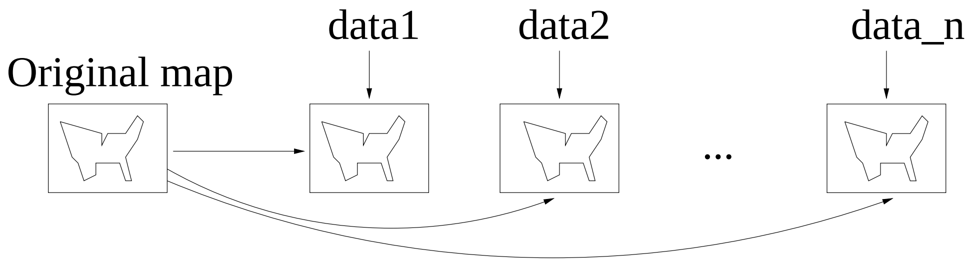


Figure 3.1: Constructing cartogram snapshots using Parallel Method

Serial Method: The *Parallel Method* provides one possible way to compute a series of cartogram snapshots. In many cases (such as a series data of annual population statistics, annul gross state product, etc.), the data change smoothly from one snapshot to the next, and thus one cartogram should be much more similar to the next cartogram in the series than to the original geographic map. This idea leads to the *Serial Method* to construct a series of cartograms.

In the *Serial Method*, the cartograms are constructed in sequence from the previous cartogram (except for the first cartogram, which is constructed from the original geographic map), i.e., instead of taking the original geographic map as the input for the transformation process, we can take the previous cartogram snapshot in the series as the input for the transformation process. Since for each step the input map is much similar to the destination cartogram, it will be computed faster and the display will be smoother. An illustration for the *Serial Method* is shown in Figure 3.2.

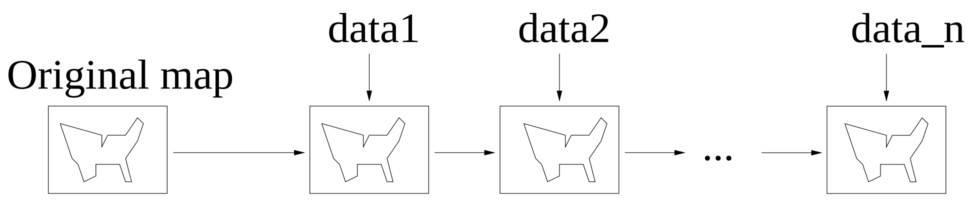


Figure 3.2: Constructing cartogram snapshots using Serial Method

Hybrid Method: This method combines the previous two methods. It generates each $k \times n$th snapshot from the original map, and all the other snapshots from the previous snapshots (see Figure 3.3).

Comparison: The Parallel Method is easily parallelized because each snapshot can be computed independently, while the Serial Method is not immediately parallelizable. However, the Parallel Method is slower than the Serial Method in a single processor computer because there is usually a bigger difference between the original map and a snapshot than between two consecutive snapshots, hence calculating the

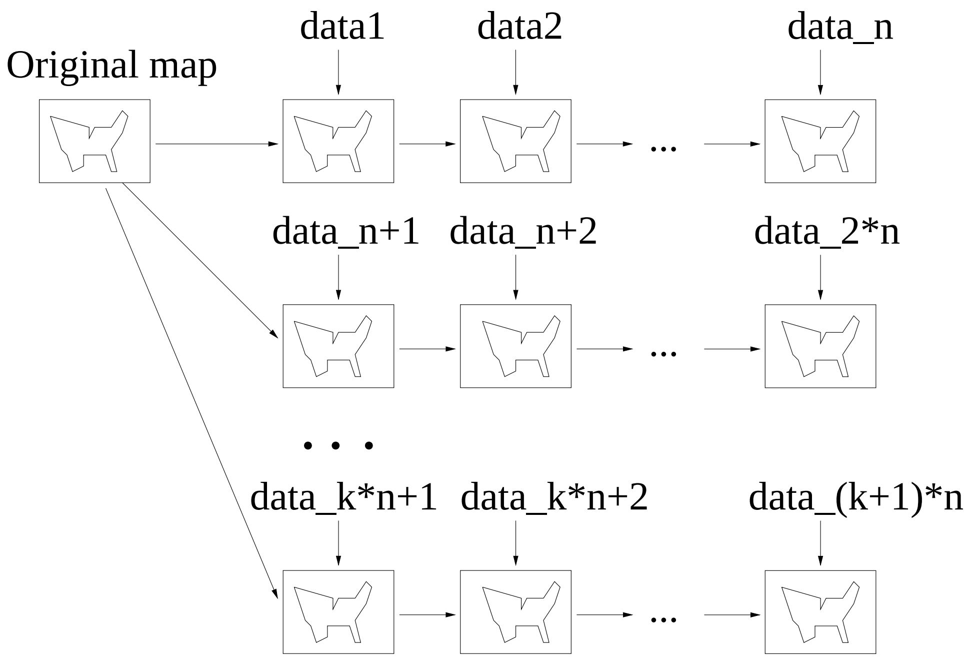


Figure 3.3: Constructing cartogram snapshots using Hybrid Method

snapshot from the original map requires generally more time than calculating it from the previous snapshot.

For the Serial Method, the cartogram snapshot quality is not as good as in the Parallel Method. Because in the Serial Method, the previous snapshot will transmit cell distortion to its successor and hence cell shape distortion may accumulate.

The Hybrid Method overcomes the cell shape distortion accumulation problem inherent in the Serial Method, while being almost as fast as the Serial Method. Table 3.1 summarizes the discussions on three methods.

Category	**Parallel Method**	**Hybrid Method**	**Serial Method**
Speed	Slow	Fast	Fast
Easily Parallelizable	Yes	Yes	No
Shape Distortion Accumulates	No	No	Yes

Table 3.1: Comparison of three Snapshots Construction Methods

3.1.2 A New Value-by-Area Cartogram Algorithm

From 1973, several computer algorithms have been developed for constructing continuous value-by-area cartograms [7, 11, 13, 30, 31]. In these algorithms, a value-by-area cartogram algorithm takes the geographic map as the initial map, divides it into many small parts (cells), and in an iterative process, enlarge or shrink each cell by some "density" value related to that cell, at the end of iteration, a value-by-area cartogram is created with each cell area error less than some threshold.

Among previous value-by-area cartogram algorithms the rubber-sheet based algorithms of [7, 11] produce the most accurate cartograms although they may be slow. We review and improve the speed of these algorithms below.

In both rubber-sheet transformations [7, 11] the map is divided into small cells (polygons). Each cell has a "value" which describes the size of its desired area. The cell is inflated (if its actual area is smaller than the desired area) or deflated (if its actual area is larger than the desired area). An iterative process to inflate or deflate the cells is done until the difference between the actual cell area and the desired cell area is less than some error tolerance value.

In order to prevent holes in the map or overlaps among the cells, in [7, 11] the inflation/deflation of any cell influences all the corner vertices of the cells in the map. The two algorithms calculate slightly differently the influence values, but in both the further the corner vertex is from the center of the inflated/deflated cell the less it is influenced or changed. When enlarging/shrinking a cell, a corner vertex is moved along the line that connects the cell center and this vertex, as shown in Figure 3.4.

Suppose that a map with area S is divided into n cells $C_1, C_2, \cdots, C_n$, the area of cell C_i is AC_i. For each cell C_i, there is a value V_i related to this cell. Then we define the average cell density D_{AVG}, desired cell area AD_i and cell distortion Δ_i for cell C_i as following:

$$D_{AVG} = \frac{\Sigma_{i=1}^{n} V_i}{S}$$

$$AD_i = \frac{V_i}{D_{AVG}}$$

$$\Delta_i = \frac{|AD_i - AC_i|}{AD_i}$$

The move distance of a corner vertex under a cell inflation/deflation is calculated in the following way:

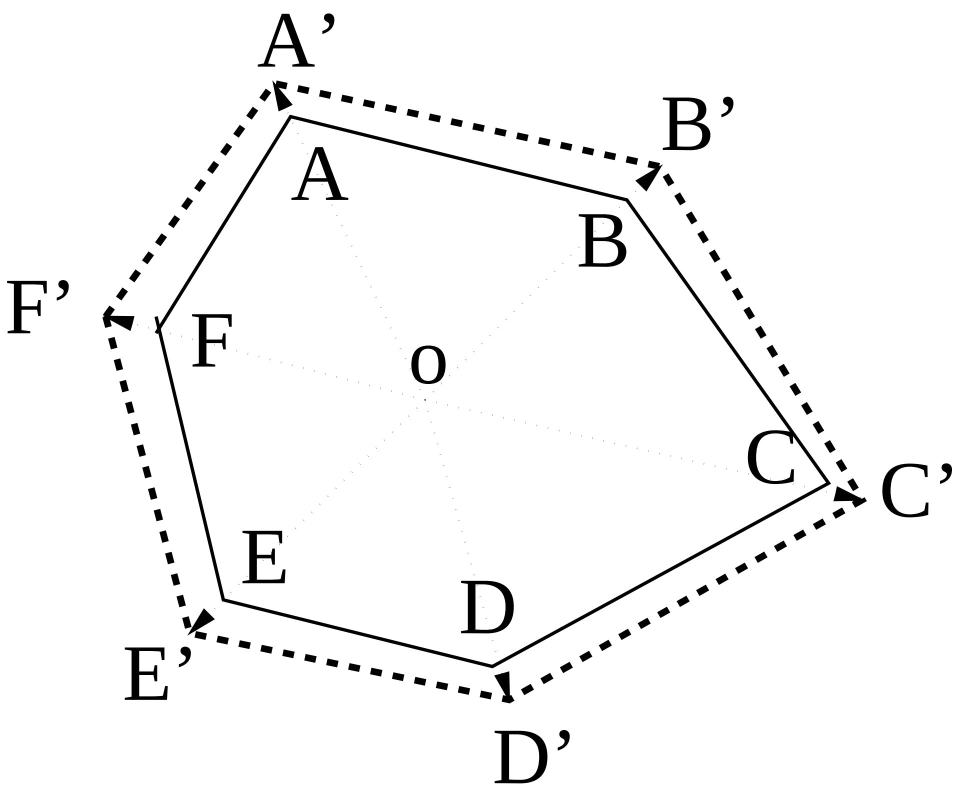


Figure 3.4: Corner Vertices Movement when Enlarging a Cell

Suppose that a cell to be inflated/deflated has area AC_i, it's desired area is AD_i. We calculate the "radius" r of the cell to be $r_0 = \sqrt{AC_i/\pi}$. Suppose that the distance between a vertex and this cell's center is r, then the move distance d is calculated as:

$$d = \begin{cases} (\frac{AD_i}{AC_i} - 1) * r_0 & if\ r \leq r_0 \\ \frac{AD_i - AC_i}{2\pi r} & if\ r > r_0 \end{cases} \tag{3.1}$$

A naive cartogram algorithm, as in [7, 11], works in the following way:

A NAIVE VALUE-BY-AREA CARTOGRAM ALGORITHM

```
Input:  A map with $n$ cells and a value for each cell,
        cell area percent error tolerance $\epsilon$,
Output: New coordinates for cell vertices that
        make them form a value-by-area cartogram
begin
  for each cell, calculate $AC_i$ and $AD_i$, begin
    repeat
    for each cell $C_i$, begin
      compute its $AC_i$
      update Cell $C_i$'s corner vertices
      update all other corner vertices
    end for
    until for all cell i, $\frac{|ACi-ADi|}{ADi} \leq \epsilon$
end
```

Note that the influence of a particular cell's inflation/deflation on the corner vertices in the map decreases with the distance from the cell center to this corner vertex and increases with the percent the cell is inflated/deflated. For example, suppose that a map with left bottom $(0, 0)$ and right top $(100, 100)$ is divided into

100×100 equal size square cells. If the cell with left bottom $(50, 50)$ and right top $(51, 51)$ inflates 1%, then the influence on a vertex say at $(52, 53)$ is about 0.0049, while the influence on a vertex at $(1, 2)$ is about 0.0001, which may be too small for consideration.

In the computation of inflation/deflation influences on the corner vertices of cells, instead of computing each influence, it is practical to ignore some small influences. Therefore, for any particular cell inflation/deflation, we can have an effective range. If a corner vertex is out of a cell inflation/deflation effective range, the change on this vertex will be too small for consideration and can be ignored.

For example, in Figure 3.5 the dotted lines represent the original cell division and $S0$ is inflated. Here the effective range is shown by the dashed cycle. The corner vertices within the effective range are changed resulting in the new cell division shown in solid lines.

In the value-by-area cartogram transformation, if a cell $S0$ with center coordinate (x_0, y_0) inflates/deflates, then instead of computing the distances from (x_0, y_0) to each of the corner vertices (x_i, y_i) in the map to see if (x_i, y_i) is inside the effective range, we can simply compute the distance d_i, which is the distance from (x_0, y_0) to the center of cell S_i. We can use the distance d_i to approximately represent the distances from (x_0, y_0) to each of the corner vertices of cell S_i in deciding if a corner vertex of cell S_j is inside the effective range. This will decrease the computational time if each cell has a lot of corner vertices.

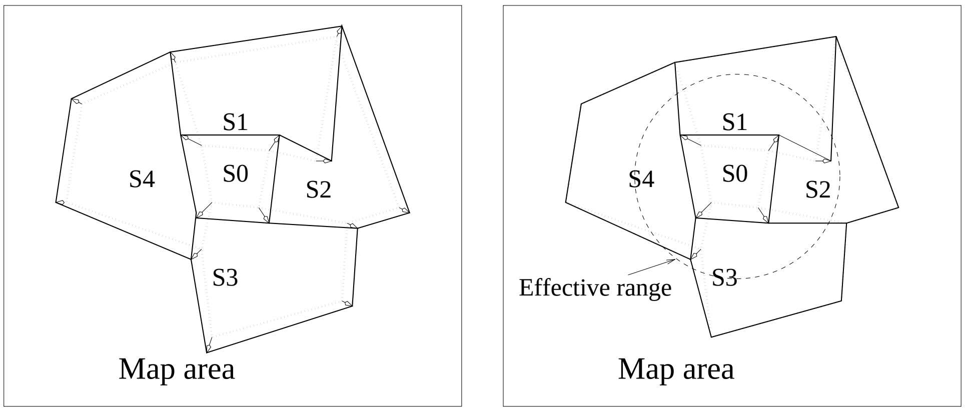


Figure 3.5: Effective Range

The following is the pseudo-code for the new algorithm.

VALUE-BY-AREA CARTOGRAM ALGORITHM

```
Input:   A map with $n$ cells and a value for each cell,
         cell area percent error tolerance $\epsilon$,
Output:  New coordinates for cell vertices that
         make them form a value-by-area cartogram
begin
  for $i = 1$ to $n$ do
    Compute $AD_i$, the desired area of the $i$th cell
  end for
  repeat
```

```
for $i = 1$ to $n$ do
  Compute $AC_i$, the area of the $i$th cell
  Compute the center $(x_i, y_i)$ of the $i$th cell
  $e_i = \frac{abs(AC_i - AD_i)}{AD_i}$, the percent area error
  if $e_i \geq \epsilon$ then
    $r_{eff} = \frac{100 * abs(AD_i - AC_i)}{\sqrt{\pi AC_i}}$
    for $j = 1$ to $n$ do
      $d_j = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$
      if $d_j \leq r_{eff}$ then
        for each corner vertex $(x, y)$ in cell $j$ do
          $d = \sqrt{(x - x_i)^2 + (y - y_i)^2}$
          if $d \leq r_{eff}$ then
            if $d \leq \sqrt{AC_i/\pi}$ then
              $x = x_i + (x - x_i)\sqrt{\frac{AD_i}{AC_i}}$
              $y = y_i + (y - y_i)\sqrt{\frac{AD_i}{AC_i}}$
            else
              $x = x_i + (x - x_i)\frac{AD_i - AC_i}{2\pi d^2}$
              $y = y_i + (y - y_i)\frac{AD_i - AC_i}{2\pi d^2}$
            endif
          endif
        end for
```

```
          **endif**
        **end for**
      **endif**
    **end for**
    **for** $i = 1$ to $n$ **do**
      Recompute $AC_i$, the area of the $i$th cell
      Recompute $e_i = \frac{abs(AC_i - AD_i)}{AD_i}$, the percent area error
    **end for**
  **until** $(\forall i,\ e_i < \epsilon)$
**end**
```

At the beginning of the algorithm there is a **for** loop which computes the desired area AD_i for each cell i, followed by a **repeat** loop, which does the value-by-area cartogram transformation.

Inside the **repeat** loop are four **for** loops. The first **for** loop computes the cell area AC_i, cell center (x_i, y_i) and the precent cell area error e_i for each center. If e_i is greater than the error tolerance ϵ, then inside the first **for** loop, the second **for** loop computes the radius of the effective range r_{eff} based on AD_i and AC_i. Then the second **for** loop computes the distance d_j as described before. If d_j is less than r_{eff}, then the third loop transforms the corner vertices of cell S_j which are inside the effective range. The goal of the fourth **for** loop inside the **repeat** loop is to compute the percent cell area error between the cell area and the desired cell area for each cell. Those are used to control the termination of the algorithm. The algorithm terminates if all the percent error between cell area and desired cell area is less than the error tolerance ϵ.

3.2 Implementation Results

We implemented in Visual C++ the algorithms and did some experiments on a 450 MHz Pentium-Pro PC with 128MB memory running Windows/NT.

3.2.1 Runtime Comparisons for Different Animation Methods

Table 3.2 gives the run time results for three problems: daily mean temperature, daily temperature spread, and monthly precipitation of the United States. Four strategies are used for creating animations. ParD is the Parallel Method with Dougenik et al.'s cartogram algorithm, ParN is the Parallel Method with the new cartogram algorithm, HybN is the Hybrid Method with the new cartogram algorithm the SerN is the Serial Method with the new cartogram algorithm. The temperature and precipitation data are from the web-site of the United States National Climatic Data Center ($http://www.ncdc.noaa.org$). For daily mean temperature and daily temperature spread animation, the periods are from January 1, 1993 to December 31,1994. For monthly precipitation animation, the period is from January 1948 to December 1997.

Problem	**Snap-shots**	**Run time (seconds)**			
		ParD	**ParN**	**HybN**	**SerN**
DMT	730	343	89	44	*
DTS	730	353	136	56	*
MPR	600	287	135	50	44

Table 3.2: Computational times for different problems (DMT: daily mean temperature. DTS: daily temperature spread. MPR: monthly precipitation)
A "*" in the table means the algorithm will run into cell distortion.

Table 3.2 shows that the Hybrid Method with the new cartogram algorithm runs much faster than the Parallel Method with Dougenik et al. cartogram algorithm.

We did several experiments to see the runtime results with different divisions of the map. Generally, the more cells a map has, the more time is required for finding the value-by-area cartogram. Figure 3.6 shows the relationship of the run time and the number of cells.

The average number of edges for the polygonal cells will also affect the computation runtime, but generally is not as much as the number of cells. Figure 3.7 shows the relationship between the animation time and the average number of

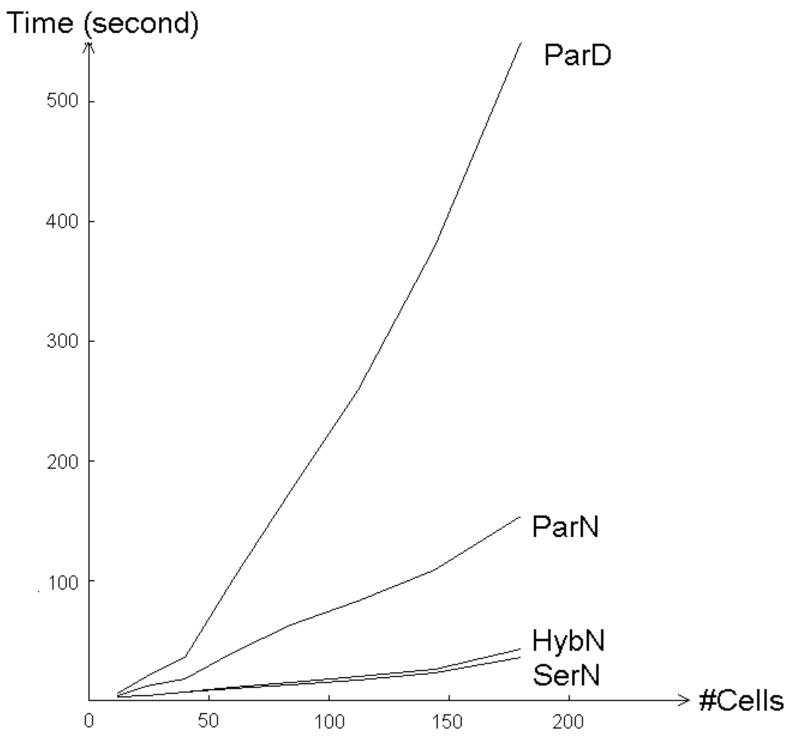


Figure 3.6: Runtime for different number of cells

edges for cells when the number of cells dividing a map is fixed.

Figures 3.6 and 3.7 together suggest experimentally that Dougenik et al.'s algorithm has an average case complexity of $O(m \times n^2)$ while our cartogram algorithm has approximately $O(m \times n)$ complexity, where n is the number of cells and m is the average number of edges for the cells. In particular, Figure 3.7 shows that they all grow linear in m, while Figure 3.6 shows that Dougenik et al.'s algorithm grows quadratically while our cartogram algorithm grows approximately linearly in n.

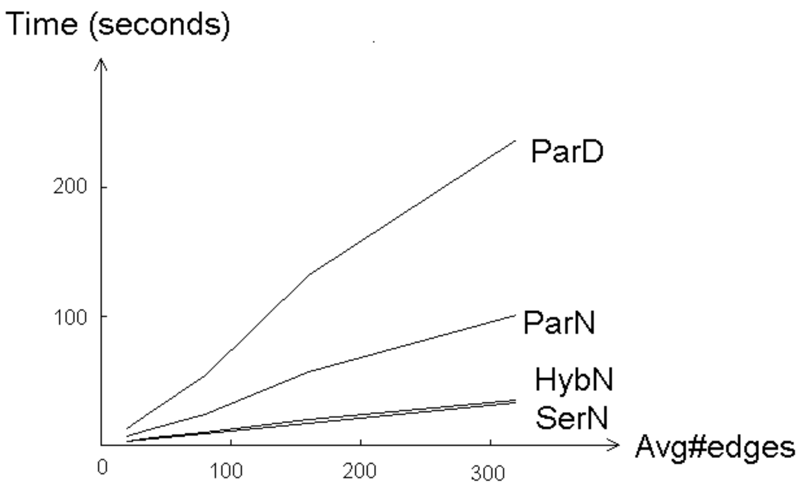


Figure 3.7: Runtime for different number of average number of cell edges

3.2.2 Accuracy of the Algorithms

Dougenik et al.'s algorithm runs slowly. The Serial Method runs fast but due to shape distortion accumulation it gives poor cartogram snapshots, or, even totally distorts the cartogram snapshots during the animation. Thus neither of them is suitable for animation.

The Parallel and the Hybrid Methods with new cartogram algorithm give highly accurate cartograms and run fast. More specifically, the Hybrid Method often runs faster and gives more accurate cartogram snapshots than the Parallel Method. Figure 3.8 shows the error comparison for the Parallel Method and the Serial Method during daily mean temperature animation.

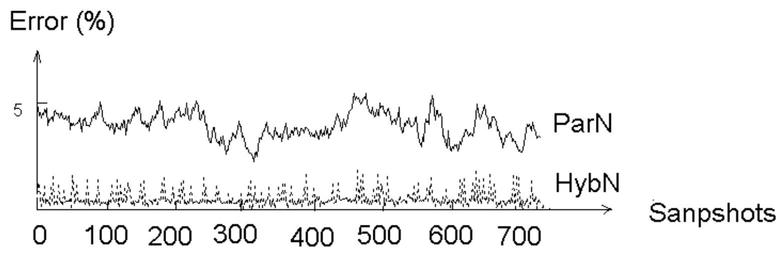


Figure 3.8: Area error comparison for Parallel Method and Hybrid Method

Chapter 4

Piecewise Linear Approximation for Time Series Data

In this chapter, We discuss an efficient way to represent and approximate GIS time series data. The presentation below is based on our work in [4, 5, 26].

Many databases contain spatiotemporal data that change continuously with time but are measured and recorded only sporadically. For example, population and various other census data in the United States is recorded only every ten years. Different weather and environmental stations throughout the world may be measuring and reporting data like air temperature, precipitation, wind direction, wind speed and levels of different air or water pollutants with different frequencies and regularities. These data are all time series data.

A time series S is a sequence of data points $(t_1, y_1), \cdots, (t_n, y_n)$ where the ts are monotone increasing time values. Note that in a time series the ts need not always increase uniformly with the same increment. Many GIS databases contain time series data, for example, the temperature and precipitation data.

It is obvious that all these spatio-temporal data cannot be available for all locations at all times. If we are interested in the value of a spatio-temporal variable at a particular time, then we have to somehow approximate that value based on some interpolation from the available data.

The interpolation could be done at two different levels. One approach is to represent the measured data in a standard relational database. Then the relational database can be embedded in a high-level computer program that retrieves the measurements, interpolates them and does other calculations. This approach may be a workable one for some scientists who are advanced programmers or who have such help readily available. It is not feasible for average users.

An alternative approach, that we advocate in this chapter, is to perform the interpolation at the time of the data entry, that is, the data should be stored as a constraint database [14, 23, 33], where the constraints are parametric functions of time that interpolate the data. This approach is advantageous because it is possible to build powerful database systems (for example, CCUBE [3], DEDALE [10] and MLPQ [28]) that can be queried by standard relational database query languages, such as relational algebra, SQL and Datalog. This enables a much wider range of users to use the database.

Applications of constraint database systems were until now severely limited to a few well-understood areas of constraint representation, for example, GIS where convex polygonal areas were represented as conjunctions of linear inequality, i.e., half-plane intersection, constraints. Our work on interpolation functions as a natural source of constraint data opens up a range of uses of constraint databases beside these narrow focus applications.

The following is an example of the time series data .

Example 4.0.1 The recording of temperature at a weather station is a time series. For example, for weather station 1 a time series may be $(0, 75), (1, 77), (2, 86), (3, 87), (4, 90)$. This can be represented by the first five tuples of relation $Temperature$ as shown in Table 4.1. For other weather stations (2,3 and 4) we can represent similarly their corresponding time series by adding more tuples to the $Temperature$ relation.

Piecewise Linear Approximation: We can see that the $Temperature$ relation can get quite large. In our system we can compress this relation by using a *piecewise linear* function. A piecewise linear function is a continuous function that is the union of a set of linear functions whose domains are disjoint. For example, the following function f is a piecewise linear function, which is the union of the linear function $2t+75$ when $t \leq 1$, the function $9t+68$ when $1 < t \leq 2$, and the function

SN	**Temp**	**t**
1	75	0
1	77	1
1	86	2
1	87	3
1	90	4
2	70	0
2	72	1
2	75	2
2	80	3
2	85	4
3	80	0
3	86	1
3	81	2
3	80	3
3	78	4
4	85	0
4	83	1
4	81	2
4	78	3
4	76	4

Table 4.1: The Temperature relation

$2t + 82$ when $t > 2$.

$$f(t) = \begin{cases} 2t + 75 & t \leq 1 \\ 9t + 68 & 1 \leq t \leq 2 \\ 2t + 82 & t \geq 2 \end{cases} \tag{4.1}$$

For a sequence of time series data, we can use the piecewise linear function to approximate these data (piecewise linear approximation).

4.1 Piecewise Linear Approximation Algorithm

Given a time series S and an error tolerance constant Ψ, the piecewise linear approximation problem is the problem of finding a piecewise linear function f such that:

$$|f(t_i) - y_i| \leq \Psi \text{ for each } (t_i, y_i) \in S. \tag{4.2}$$

and

$$\text{The corner vertices of } f \text{ must occur in } S \tag{4.3}$$

In general, the smaller Ψ is the more pieces the piecewise linear function will contain.

Piecewise linear approximation is a traditional problem in applied mathematics. There are some algorithms that transform a sequence of time series data into a piecewise linear function where the number of pieces in the piecewise linear approximation is minimized. Hakimi et. al. [12] give an $O(N^2)$ (N is the number of points in S) algorithm which computes such an approximation. The complexity is improved to $O(N^{\frac{4}{3}+\delta})$ (where δ can be any positive value) by Agarwal et. al. [1].

There are some other faster algorithms which are not optimal (the number of pieces in the approximation is not minimal), such as Ramer's algorithm [25] and David et. al.'s algorithm [8]. If a sequence of time series data has N points and is approximated with N' pieces, then the complexity of Ramer's algorithm and

David et. al.'s algorithm is $O(N \log N')$.

If a time series data set S contains a large number of data points, then the above algorithms may run slow. Another disadvantage for these algorithms is that each time even if we slightly modify or add only one point, we have to recompute the approximation based on the whole data set. In geographic databases, adding new data may be very frequent, making these algorithm very inefficient. In this case, we may need a faster approximation algorithm, although it may not give the result with optimal number of pieces in the approximation. Below we present such a piecewise linear approximation algorithm that runs in $O(n)$ worst case time and is very efficient to append new points in the database.

We will describe our algorithm below. First let us define some terms to be used in explaining the algorithm.

Definition 4.1.1 On the time interval $[t_b, t_e]$, let us define $Y_{b,e}(t)$ to be the linear function:

$$Y_{b,e}(t) = \frac{y_e - y_b}{t_e - t_b}(t - t_b) + y_b \tag{4.4}$$

Note: The linear function $Y_{b,e}(t)$ can be drawn as a line segment with end-points (t_b, y_b) and (t_e, y_e).

Definition 4.1.2 Given two points (t_b, y_b) and (t_e, y_e) where $(b < e)$, and the maximum approximation error threshold is Ψ, then the *lower line* for these two points is the line which passes through the points (t_b, y_b) and $(t_e, y_e - \Psi)$, denoted as:

$$L_{b,e}(t) = \frac{(y_e - \Psi) - y_b}{t_e - t_b}(t - t_b) + y_b \tag{4.5}$$

and the *upper line* for these two points is the line which passes through the points (t_b, y_b) and $(t_e, y_e + \Psi)$, denoted as:

$$U_{b,e}(t) = \frac{(y_e + \Psi) - y_b}{t_e - t_b}(t - t_b) + y_b \tag{4.6}$$

Note: The lower line $L_{b,e}(t)$ can be drawn as a line that has one endpoint (t_b, y_b) and passes through $(t_e, y_e - \Psi)$. Similarly, the upper line $U_{b,e}(t)$ can be drawn as a line that has the same endpoint (t_b, y_b) and passes through $(t_e, y_e + \Psi)$.

Our piecewise linear approximation algorithm is shown below. The algorithm initializes the values of b and e to be 1 and 2. Note that the line segment with endpoints $(t_1, y_1), (t_2, y_2)$ is the smallest possible first piece of the piecewise linear function.

In the **if** clause of the **while** loop, the algorithm creates a new piece $Y_{b,e}(t)$ if the sequence $(t_b, y_b), \ldots, (t_e, y_e)$ can be approximated but the one longer sequence

$(t_b, y_b), \ldots, (t_{e+1}, y_{e+1})$ cannot be approximated as required by Formula (4.2).

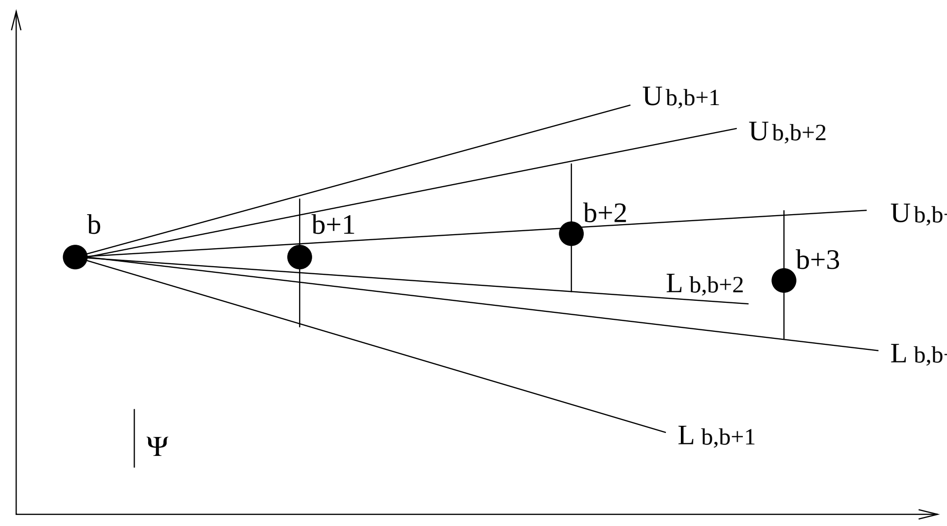


Figure 4.1: Successive change of U and L

In the **else** clause of the **while** loop, the algorithm extends the current sequence by one point and if necessary tightens both the current lower and the current upper lines. The working of the **else** clause is illustrated in Figure 4.1. There we see three pairs of lower and upper lines. Suppose that we enter three times successively the **while** loop and each time execute the **else** clause. Then the lower line L will after the first iteration $L_{b,b+1}$, in the second $L_{b,b+2}$, and in the third $L_{b,b+2}$. Similarly, the upper line U will be after the first iteration $U_{b,b+1}$, in the second $U_{b,b+2}$, and in the third $U_{b,b+3}$. Note that we get the highest slope lower bound line and the smallest slope upper line as the final result for L and U, respectively.

PIECEWISE LINEAR APPROXIMATION ALGORITHM:

Input: A time series S with n pairs $(t_1, y_1), \ldots, (t_n, y_n)$.

Ψ the maximum error threshold in the approximation.

Output: A piecewise linear function approximation of S.

Local Vars: The b and e are integer variables such that the sequence $(t_b, y_b), \ldots, (t_e, y_e)$ can be approximated by one piece.

L and U stand for the current lower and upper lines.

```
$b := 1$ and $e := b + 1$
$L := L_{b,e}$ and $U := U_{b,e}$

while $e < n$ do
   if $y_{e+1} < L(t_{e+1})$ or $y_{e+1} > U(t_{e+1})$ then
      Create a piece $Y_{b,e}$ defined by Formula (4.4)
      $b := e$ and $e := b + 1$.
      $L := L_{b,e}$ and $U := U_{b,e}$.
   else
      if $L_{b,e+1}(t_{e+1}) > L_{b,e}(t_{e+1})$ then
         $L := L_{b,e+1}$
```

```
        end-if

        if $U_{b,e+1}(t_{e+1}) < U_{b,e}(t_{e+1})$ then
            $U := U_{b,e+1}$
        end-if
        $e := e + 1$
    end-if
end-while

Create a piece $Y_{b,e}$ defined by Formula (4.4)
```

Example 4.1.1 Suppose the maximum error threshold $\Psi = 3$, and given the relation as Table 4.1 from a relational database. The piecewise linear approximation algorithm will accept this relation as input, and transform it to a piecewise linear function for each weather station (that is, for each sn). The resulting constraint relation is shown as Table 4.2.

The representation for this is shown in Table 4.2.

The piecewise linear approximation algorithm first transforms the relational

SN	**Temp**	**t**	
sn	$temp$	t	$sn = 1,\ temp = 2t + 75,\ t \geq 0,\ t \leq 1.$
sn	$temp$	t	$sn = 1,\ temp = 9t + 68,\ t > 1,\ t \leq 2.$
sn	$temp$	t	$sn = 1,\ temp = 2t + 82,\ t > 2,\ t \leq 4.$
sn	$temp$	t	$sn = 2,\ temp = 3.75t + 70,\ t \geq 0,\ t \leq 4.$
sn	$temp$	t	$sn = 3,\ temp = 6t + 80,\ t \geq 0,\ t \leq 1.$
sn	$temp$	t	$sn = 3,\ temp = -2.67t + 88.67,\ t > 1,\ t \leq 4.$
sn	$temp$	t	$sn = 4,\ temp = -2.25t + 85,\ t \geq 0,\ t \leq 4.$

Table 4.2: The Temperature Relation

tuples whose $sn = 1$ into a piecewise linear function. There are 5 tuples for $sn = 1$, therefore, $n = 5$. And it gets the time-value pairs $(t_1 = 0,\ y_1 = 75)$, $(t_2 = 1,\ y_2 = 77)$, $(t_3 = 2,\ y_3 = 86)$, $(t_4 = 3,\ y_4 = 87)$ and $(t_5 = 4,\ y_5 = 90)$.

First it intializes the b to 1, and e to 2, then sets the current upper line to be the line $L_{1,2}$, and the current upper line to be the line $U_{1,2}$. Then, since e is 2 which is smaller than n, it enters the **while** loop. For the top level **if** clause, the condition $y_{e+1} < L(t_{e+1})$ or $y_{e+1} > U(t_{e+1})$ is true. This is because the condition is equivalent to the condition $86 < L(2)$ or $86 > U(2)$. $L(2)$ is $L_{1,2}$, that is, $y = -t + 76$ by its definition, so $L(2) = -2 + 76 = 74$. $U(2)$ is $U_{1,2}$, that is, $y = 5t + 70$ by its definition, so $U(2) = 5 \times 2 + 70 = 80$. Therefore, the condition $86 < L(2)$ or $86 > U(2)$ is true. So it enters this clause. Here it creates a piece $Y_{1,2}$ which is $2t + 75$, then sets the b to 2, and e to 3, and updates the current L to be $L_{2,3}$ and U to be $U_{2,3}$. Then since current e is 3 which is less than n, it enters the **while** loop again.

The condition in the top level **if** clause is still true, so it enters this clause. Then it creates a piece $Y_{2,3}$ which is $9t + 68$, then sets the b to 3, and e to 4, and updates the current L to be $L_{3,4}$ and U to be $U_{3,4}$. Then since current e is 4 which is less than n, it enters the **while** loop again.

At this time the condition in the top level **if** clause is not true, so it enters the **else** clause. Here it updates the current lower line L to be $L_{3,5}$ since the condition $L_{b,e+1}(t_{e+1}) > L_{b,e}(t_{e+1})$ is true, and updates the current upper line U to be $U_{3,5}$ since the condition $U_{b,e+1}(t_{e+1}) < U_{b,e}(t_{e+1})$ is also true. Then increase e by 1, that is, $e = 6$. Because now e is greater than n, it exits from the **while** loop.

Then it executes the last statement, so it creates the last piece $Y_{3,5}$, which is $2t + 82$.

Similarly, the peicewise linear approximation algorithm transforms the tuples for other weather stations into corresponding piecewise linear functions. The final transformed constraint relation is shown as Table 4.2.

Theorem 4.1.1 The piecewise linear approximation algorithm is correct for any error tolerance value Ψ and time series S.

Proof: We have to show that for any S and Ψ the algorithm finds a piecewise linear function f that satisfies Formula 4.2. To show that, it is enough to prove the following invariant condition for each entry of the **while** loop, by induction on the number of entries:

Invariant Condition (1): The line segment $Y_{b,e}(t)$ on the time iterval $[t_b, t_e]$ satisfies Formula 4.2.

For the first entry of the **while** loop, the initialization implies the line segment $Y_{1,2}$. Clearly, that satisfies Formula 4.2 because it gives zero error for both t_1s and t_2.

Now we assume that invariant condition (1) holds before some entry of the **while** loop, and then we show that it will also hold before the next entry or exit from the **while** loop.

There are two basic cases. The first case is when we enter the **if** clause of the **while** loop. In that case we add the current $Y_{b,e}(t)$ to the piecewise linear function, and that is correct. Then we reset the values of b and e to be two consequtive points. Therefore, similarly to the initialization, these will also satisfy Formula 4.2.

For the **else** clause in the **while** loop, we prove the following condition that holds before entering the else clause.

Invariant Condition (2): If $f(t)$ is a line that passes through (t_b, y_b) and is between L and U (i.e., has a higher slope than L but a smaller slope than U has), then $|f(t_i) - y_i| \leq \Psi$ for each $b \leq i \leq e$.

We fix the value of b and prove the condition by induction on $e - b$. (This corresponds to repeatedly executing the **else** clause.)

If $e = b + 1$, then condition (2) obviously holds.

Now assume that the condition holds for some $k = e - b$. Then we prove it for $k + 1$. Clearly k can increase only if e has increased by one since we fixed the value of b.

If we increased e since the last entry of the **else**, then we must have also updated L and U. Let the previous values be L' and U' and the new values be simply L and U. Since the slope of L is greater than or equal to that of L', and the slope of U is less than or equal to that of U', there can be only a smaller or equal region between L and U than between L' and U'.

Let $f(t)$ be any line that passes through (t_b, y_b) and is between L and U. Then by the above, $f(t)$ also is between L' and U'. Therefore, by the induction hypothesis $|f(t_i) - y_i| \leq \Psi$ holds for $b \leq i < e$. Also, by the definition of L and U, we have that $|f(t_e) - y_e| \leq \Psi$. This proves that invariant condition (2) must hold.

Finally, note that invariant condition (2) implies invariant condition (1). That is because if we are in the **else** clause, then we can choose for $f(t)$ the linear function $Y_{b,e}(t)$.

Finally, if we exit the while loop, then we get the last piece, which also must satisfy Formula 4.2 because none of our arguments above depended on the value of n. Hence if we had more values we could continue by entering again the **while** loop, that is, invariant condition (1) still must hold. ■

Next we analyze the computational complexity of our approximation algorithm.

Theorem 4.1.2 The computational complexity of the piecewise linear approximation algorithm is $O(n)$ in the worst case where n is the number of points in the time series to be approximated.

Proof: There is only loop, the **while** loop in the piecewise linear approximation algorithm. The **while** loop is executed only at most $n-2$ times, because initially the value of e is two, then it is incremented by one in each iteration until $e = n$.

We also have to show that each iteration of the **while** loop takes only a constant time. Within the **while** loop the top level **if** statement has two clauses. The **then** clause takes a constant time, because there we only do a fixed number of comparison and assignment operations and add one piece to the piecewise linear function, which will be the output of the algorithm. By keeping the pieces in a linked list and a pointer to the end of the last list, we can do the addition in constant time. In the **else** clause we again do only fixed number of comparison and assignment operations.

Therefore, the worst case computational complexity of this algorithm is $O(n)$.

■

Remark: There are other advantages of the piecewise linear approximation algorithm beside data compression. First, we do interpolation as well as conversion of data. For example, this allows us to ask what was the temperture at time instances other than the original time instances when the original time series measurements were taken. Second, because the data is much smaller, querying and visualizations could be also done faster using the compressed data.

4.2 Analysis of the Expected Number of Points in a Piece

Now we analyze the expected number of this ratio, assuming that the time series $(t_1, y_1), \cdots, (t_n, y_n)$ satisfies the following property, for some constant M and for each $1 < i \leq n$

$$\begin{cases} y_1 = 0 \\ Prob(y_i - y_{i-1} = M) = 0.5 \\ Prob(y_i - y_{i-1} = -M) = 0.5 \end{cases} \tag{4.7}$$

For example, consider the time series which starts with $(0, 0)$ and records for each later time a coin is flipped and the number of heads minus the number of tails seen since the beginning. This time series satisfies Property (4.7) with $M = 1$ if heads and tails have the same probability.

As another example, the daily temperature could be described by a time series that satisfies Property (4.7), if we use a thermometer in which the adjacent scales are M Fahrenheit degrees apart instead of the usual single Fahrenheit degrees, where M is the largest daily change, and if we record only on those days when there is a change in temperature according to the rougher thermometer.

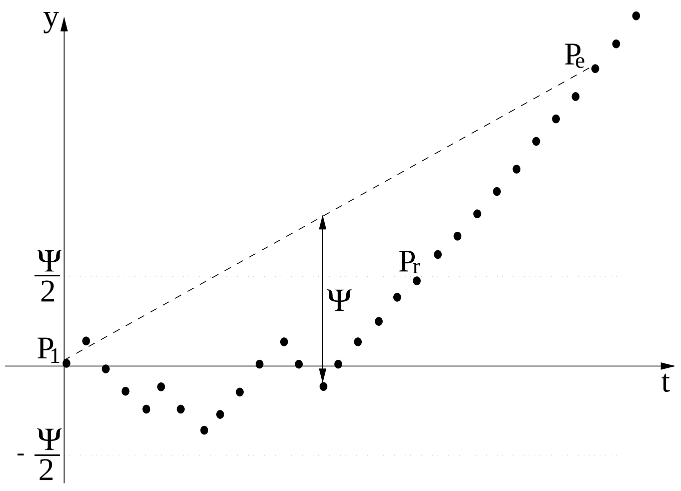


Figure 4.2: Piecewise Linear Approximation for Points

Let $E(\Psi, M)$ be the expected number of original points spanned by a single piece of the piecewise linear approximation, including the two endpoints, when the approximation uses the tolerance Ψ and the time series satisfies Property (4.7). We can prove the following.

Theorem 4.2.1 If a time series satisfies Property (4.7), then

$$E(\Psi, M) \geq \left(\left\lfloor \frac{\Psi}{2M} \right\rfloor + 1 \right)^2 .$$

Proof: Let $R(h)$ be the expected number of additional points before the time series goes out of the error range $-\frac{\Psi}{2} \leq y \leq \frac{\Psi}{2}$ when it starts from height hM as shown in Figure 4.2. For $k = \lfloor \frac{\Psi}{2M} \rfloor$, we have the following system of equations:

$$\begin{cases} R(-k) = & \frac{1}{2}(1 + R(-k+1)) \\ R(i) = & \frac{1}{2}(1 + R(i-1)) + \frac{1}{2}(1 + R(i+1)) \quad \text{for } -k < i < k \\ R(k) = & \frac{1}{2}(1 + R(k-1)) \end{cases} \tag{4.8}$$

Note that $-kM$ is the lowest height that is still within the error range. The first equation is true because there is one half chance that we go out of the error range and also one half chance that we go to height $(-k+1)M$ and continue from there. The other two equations can be similarly explained.

Next we will prove that the solution for the above equations is:

$$R(i) = k^2 - i^2 + 2\Psi.$$

We will prove by induction.

1. We prove that $R(k) = k^2 - k^2 + 2\Psi$ and $R(k-1) = k^2 - (k-1)^2 + 2\Psi$.

   We sum all the equations in 4.8, then we can get the following equation:

   $$\frac{1}{2}(R(k) + R(-k)) = 2\Psi$$

   Because $R(k)$ and $R(-k)$ are symmetric in equations 4.8, they have the same value, hence we can get that:

   $$R(k) = 2\Psi = k^2 - k^2 + 2\Psi$$

Having the value of R_k, from the equation

$$R(k) = \frac{1}{2}(1 + R(k-1))$$

We can get

$$R(k-1) = k^2 - (k-1)^2 + 2\Psi$$

2. Suppose that $R(i) = k^2 - i^2 + 2\Psi$ and $R(i-1) = k^2 - (i-1)^2 + 2\Psi$ for some $1 \le i \le k$,

   Then from the equation

$$R(i-1) = \frac{1}{2}(1 + R(i)) + \frac{1}{2}(1 + R(i-2))$$

   We have

$$\begin{aligned} R(i-2) &= 2R(i-1) + R(i) - 2 \\ &= 2k^2 - 2(i-1)^2 + 4\Psi - K^2 + i^2 - 2\Psi - 2 \\ &= k^2 + 2\Psi - (i^2 - 4i + 4) \\ &= k^2 - (i-2)^2 + 2\Psi \end{aligned}$$

From 1 and 2 we can get that

$$R(i) = k^2 - i^2 + 2\Psi.$$

for all $0 \leq i \leq k$. For the cases that $-k \leq i \leq 0$, we can similarly prove that

$$R(i) = k^2 - i^2 + 2\Psi.$$

Hence,

$$R(0) = k^2 + 2\Psi.$$

Now suppose that the piecewise linear approximation algorithm creates a piece between P_1 and P_v as shown in Figure 4.2. If we connect P_1 to any point before P_u, then the error will not exceed Ψ because all points lie between $-\frac{\Psi}{2}$ and $\frac{\Psi}{2}$. Hence the approximation algorithm can go at least to point P_u. This shows that $E(\Psi, M) \geq R(0) + 1 = (k+1)^2 = \left(\left\lfloor \frac{\Psi}{2M} \right\rfloor + 1\right)^2$. Hence the theorem holds. ■

For example, for the coin flipping time series when $\Psi = 6$ each piece of the piecewise linear approximation function is expected to span at least 16 original time series points.

4.3 Update of Piecewise Linear Function

The update module is responsible for modifying the database as the user requests. Usually the query languages are augmented with special language constructs to express updates. The most important types of update operations are insertions and deletions of tuples in relational databases.

In this section we consider what happens in the system if we approximated by a constraint database some relational database that represents a time series and the user requests an insertion or deletion of a point in the time series. Note that the user can request insertions and deletions of time series points (i.e., tuples of the relational database) and not the constraint database, because the constraint database representation is hidden from the user.

If the user requests a deletion of a time series data point, then the request can be ignored because the approximation function still satisfies the error tolerance for the remaining points.

The insertion of points is much more complex. In this case, we have to update the piecewise linear function. Consider Figure 4.3. There the original piecewise linear function is shown as a solid black line in (1). In (2) the point P_1 is to be inserted, but the piecewise linear function is not changed since P_1 is in the error tolerance Ψ. In (3) the point P_2 is to be inserted, and the piecewise linear function is updated by splitting the middle piece into two pieces. In (4) the point P_3 is to be inserted, and the piecewise linear function is updated by splitting the third piece into two pieces.

The system insertion algorithm for a single data point (t_α, y_α) is shown below.

Remark 3: In Remark 2, we mentioned that a piecewise linear approximation

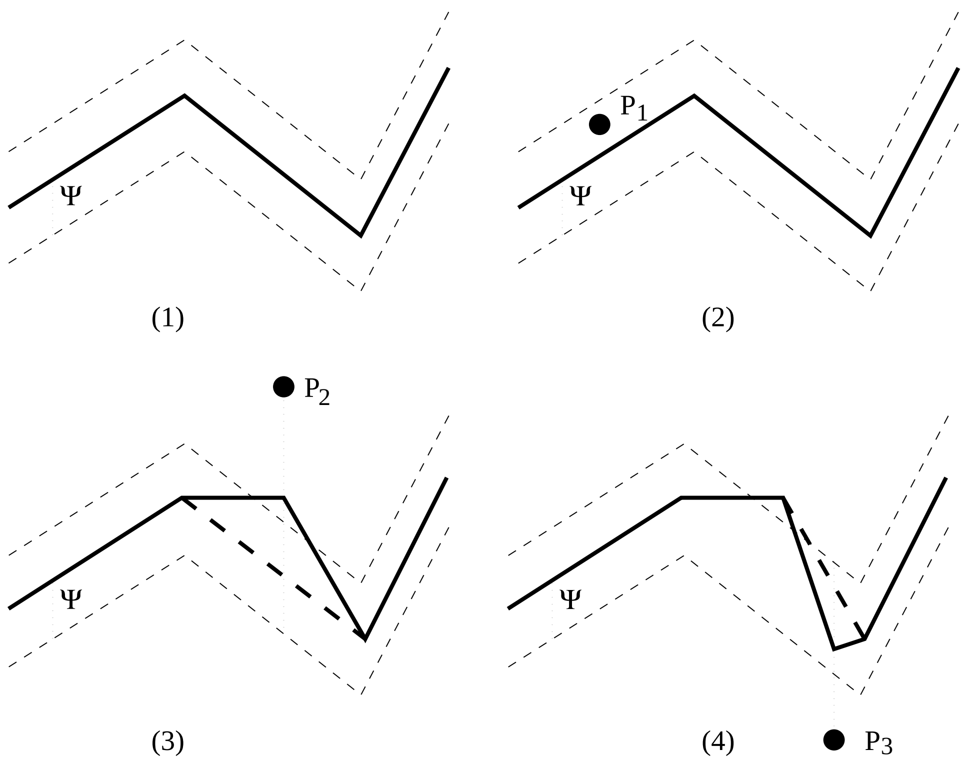


Figure 4.3: Inserting Points into Piecewise Linear Function

can be represented as a set of endpoints of the pieces. Now, we add a Boolean tag o to each point. The tag will be *true* if it is a point that is an original point, otherwise it is a point which was inserted and the tag will be *false*. This allows us to reconstruct from any updated piecewise linear function f, the original piecewise linear function, denoted f_o, as the sequence of points with the *true* tags. Finally, we assume that for no point (t_α, y_α) to be inserted is there already a point with time t_α.

INSERTION ALGORITHM:

Input: A piecewise linear function f represented as a sequence of points $(t_1, y_1, o_1), \ldots, (t_n, y_n, o_n)$ as in Remark 3.

Ψ the maximum error threshold in the approximation.

(t_α, y_α) the point to be inserted.

Output: An updated piecewise linear function.

```
if $t_\alpha < t_1$ then
  Add $(t_\alpha, y_\alpha, false)$ as the first point in $f$.
else if $t_\alpha > t_n$ then
  Add $(t_\alpha, y_\alpha, false)$ as the last point in $f$.
else if $f_o(t_\alpha) - \Psi > y_\alpha$ then
  Add $(t_\alpha, \frac{1}{2}((f_o(t_\alpha) + \Psi) + y_\alpha), false)$ between points with times $t_i < t_\alpha$ and $t_{i+1} > t_\alpha$.
else if $f_o(t_\alpha) + \Psi < y_\alpha$ then
  Add $(t_\alpha, \frac{1}{2}((f_o(t_\alpha) - \Psi) + y_\alpha), false)$ between points with times $t_i < t_\alpha$ and $t_{i+1} > t_\alpha$.
end-if
```

We can show the following theorem.

Theorem 4.3.1 Suppose that a time series S is approximated by a piecewise linear function f_o with n "pieces" and Ψ error tolerance. Then any set I of m insertions such that each insertion point is at most some constant $\delta \geq \Psi$ distance from f_o can be done by the TAQS insertion algorithm such that the updated piecewise linear function f has at most $n + m$ "pieces" and the following holds.

$$|f(t_i) - y_i| \leq \frac{\Psi + \delta}{2} \text{ for each } (t_i, y_i) \in S \cup I.$$

Proof: From the algorithm above, there are four cases to insert one point (t_α, y_α), which are (1): $t_\alpha < t_1$, (2): $t_\alpha > t_n$, (3): $t_1 < t_\alpha < t_n$ and $f_o(t_\alpha) - \Psi > y_\alpha$, and (4): $t_1 < t_\alpha < t_n$ and $f_o(t_\alpha) + \Psi < y_\alpha$.

First we prove that the updated piecewise linear function f has at most $n+m$ pieces after m insertions. In either of the four cases, the algorithm adds one point to f. In other cases, the algorithm does not add any point to f. Therefore, for a sequence of m insertions, at most m points are added to f. Further, no points are ever deleted by the insertion algorithm. Hence, f has at most $n + m$ points.

Next, we prove that $|f(t_i) - y_i| \leq \frac{\Psi+\delta}{2}$ for each $(t_i, y_i) \in S \cup I$. We prove this by induction. Let us assume that after a sequence of insertions the condition is true and now we are inserting some new point (t_α, y_α). We prove that the condition also holds after the insertion of (t_α, y_α).

Case (1): The insertion algorithm in effect adds a new piece with endpoints (t_α, y_α) and (t_1, y_1) to f. The condition is clearly true in this case, because the point (t_α, y_α) is contained in the new piece.

Case (2): This is similar to case (1).

Case (3): In this case, the old piece between t_i, y_i and (t_{i+1}, y_{i+1}) is deleted and replaced with two new pieces, one with endpoints $A(t_i, y_i)$ and $C(t_\alpha, \frac{1}{2}((f_o(t_\alpha) + \Psi) + y_\alpha)$, and the other with endpoints $C(t_\alpha, \frac{1}{2}((f_o(t_\alpha) + \Psi) + y_\alpha)$ and $B(t_{i+1}, y_{i+1})$.

Note that by the induction hypothesis, all points in S before A or after B still satisfy the condition. Hence we only have to prove that the condition is still true for the points of f that are between A and B. First let us consider the points on the piece AC.

Consider the original piecewise linear function between A and B. Let D and E be points on the line $f_o(t) - \Psi$, and F and G be points on the line $f_o(t) + \Psi$ as shown in Figure 4.4. The coordinates of these four points can be calculated to be $D(t_i, f_o(t_i) - \Psi)$, $E(t_\alpha, f_o(t_\alpha) - \Psi)$, $F(t_i, f_o(t_i) + \Psi)$ and $G(t_\alpha, f_o(t_\alpha) + \Psi)$. For the point to be inserted (t_α, y_α), we calculate the following:

$$
\begin{aligned}
&|f(t_\alpha) - y_\alpha| \\
&= \tfrac{1}{2}((f_o(t_\alpha) + \Psi) + y_\alpha) - y_\alpha \text{ by y coordinate of C} \\
&= \tfrac{1}{2}((f_o(t_\alpha) + \Psi) - y_\alpha) \\
&= \tfrac{1}{2}(\Psi + (f_o(t_\alpha) - y_\alpha)) \\
&\leq \tfrac{\Psi+\delta}{2}.
\end{aligned}
\tag{4.9}
$$

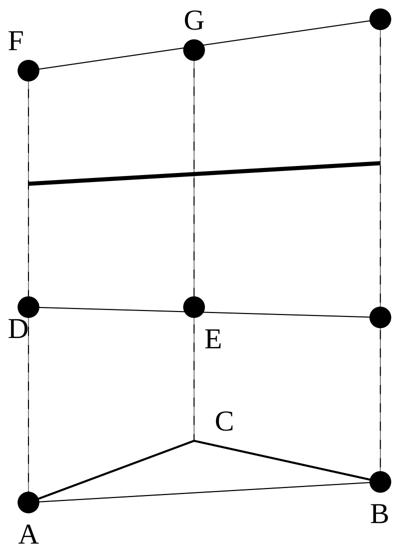


Figure 4.4: Insert a point

The last inequality follows form the condition that each point inserted is at most δ distance from the original piecewise linear function. Hence (t_α, y_α) satisfies the condition.

Note that there cannot be any other I point H before (t_α, y_α) that is between A and C and has more than Ψ distance from f_o. If we had, then we would have to use either AH or HB instead of AB when we are inserting (t_α, y_α).

Now we can assume that all S and I points before (t_α, y_α) and between A and C are at most Ψ distance from f_o. Therefore, these all fall into the trapezoid region $DEGF$, showing that any point within $DEGF$ satisfies the condition.

At first we show the condition for the corner vertices. For D and F the condition is true by the induction hypothesis, that is, they are both at most $\frac{\Psi+\delta}{2}$ distance from A. Note that y_α is at most δ and both E and G are at most Ψ distance from f_o. Since the y coordinate of C is at the midpoint of y_α and the y coordinate of G, both G and E are at most $\frac{\Psi+\delta}{2}$ distance from C.

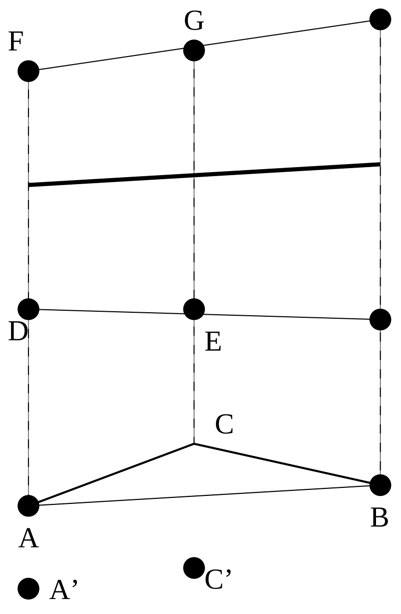


Figure 4.5: Proof condition for point M

Let A' be the point exactly $\frac{\Psi+\delta}{2}$ below F, and let C' be the point exactly $\frac{\Psi+\delta}{2}$ below G. Suppose that $M = (t, y)$ is any point within $DEGF$. Let M_1 be the point directly above M and intersecting the line segment FG and M_2 be the point directly below M and intersecting the line segment $A'C'$ as shown in Figure 4.5.

Clearly, the distance between M and AC is less than the distance between M_1 and M_2, which is exactly $\frac{\Psi+\delta}{2}$. Hence M must satisfy the condition. Therefore, all points within $DEGF$ satisfy the condition.

The above took care for points between A and C. We can prove similarly that all points in S and in I before (t_α, y_α) and between C and B also satisfy the condition.

Case (4): It is similar to Case (3). ■

For example, if $\delta = 3\Psi$, then the error tolerance for the updated piecewise linear approximation will be 2Ψ for all original and newly inserted data points.

4.4 Export Conversion

The export conversion deals with the conversion from constraint databases to relational databases. Constraints in general cannot be converted to a relational database representation because that would require an infinite number of tuples. However, if there are a finite set of time instances which are of interest, then we can generate from the constraint database a relational database that contains the values for the needed time instances. This conversion is illustrated in an example below.

Example 4.4.1 Suppose that we are given the constraint database relation $Temperature$ shown in Table 4.2 and we are interested in the temperatures at times 1 and 1.5. Then we can convert the constraint database to the relational database shown in Table 4.3.

SN	**Temp**	**t**
1	77	1
1	81.5	1.5
2	73.75	1
2	75.625	1.5
3	86	1
3	84.665	1.5
4	82.75	1
4	81.625	1.5

Table 4.3: The Conversion Result

Note that we could not obtain the above table from the relation database shown in Table 4.1 because that does not give the temperature for the time $t = 1.5$. By using the interpolation implicit in the piecewise linear approximation that actually yields the constraint database in Table 4.2, we could access an approximate value of the temperature at any time instance between 1 and 4 including 1.5. The constraint database representation is better in this case as we do not know ahead what time instances the users may be interested in.

Chapter 5

Using Piecewise Linear Approximation in Cartogram Animation

5.1 Approximate Cartogram Animation

Value-by-area cartogram animation is generally used to animate geographically distributed time series data. These data usually change continuously but are usually measured sporadically and hence we only have the data at some time instance. A simple animation strategy is to create one snapshot for each recorded time instance. However, there are some shortcomings in this simple animation strategy:

First, we can not freely choose different time granularity for animation. If the data are sampled sparsely over time, without inserting some intermediate snapshots, the animation may look “jumping” from one snapshot to a quiet differently next one.

Second, sometimes we may need to animate the values which are not in the original data set. For example, if the temperature is measured every 2 hours, that is, it is measured at 0:00am, 2:00am, $\cdots$, 10:00pm, they we can not show animation of temperature at 11:00am each day.

Third, suppose there are two temperature datasets for two cells, one of them records the temperatures every 10 days, the other one does every 7 days. How to animate both of them in the same animation? They will have same day's data for every 70 days.

Some GIS data may be very huge. For example, in the United States there are over 8,021 weather stations, if each station measures temperature and precipitation data hourly, then each year large quantities of data will be gathered. The data may need to be compressed in order to be efficiently queried and displayed.

As discussed in Section 4, piecewise linear approximation can compress a sequence of time series data, while guaranteeing some error tolerance. It is very useful to approximate time series values for animation.

Some advantages for using piecewise linear approximation to approximate the data for animation are:

1. We can select the animation time granularity to get a smooth animation.

2. We can animate the data for any time instance.

3. If the data volume is huge and accessing them requires sometime, by compressing them into piecewise linear functions and access much fewer number of these functions, it is hopeful to decrease data access time and thus have faster animation.

4. At some extent, we can use the piecewise linear approximation to do some prediction.

Piecewise linear approximation can compress the data and provide some interpolation and prediction. However, generally it will introduce some error between the real value and the approximated value. If in some cases we need to have the exact values at the recorded time instances, we can use the piecewise linear approximation and let the error tolerance to be zero.

5.2 Animation to Real Data

Next we give an example to show how we can use piecewise linear approximation in value-by-area cartogram animation.

Example 5.2.1 We would like to animate the daily mean temperature data in the continental United States with each state as a cell. There are over 8,021 weather stations in the continental United States, which report the weather data for its area. The temperature for each state is calculated as the average temperature of all weather stations inside this state.

The 8,021 weather stations record each day the high temperature, low temperature and precipitation. Each high temperature, low temperature or precipitation is recorder in a 4 bytes string. Hence, the 50 years data are 1.76GB large.

If we use piecewise linear approximation to approximate each stations temperature with error tolerance $\Psi = 10$, the data can be compressed to about 96MB. For this volume of data, retrieval can be done much faster.

Another advantage for piecewise linear approximation is the data interpolation. In the weather dataset as above, there are often some data missing for some weather stations for some day. If we are interested in the values at a particular day for all locations, we have to somehow approximate these values based on piecewise linear approximation.

Chapter 6

Approximate Query Evaluation Method

In Chapter 4, we described that time series data can be approximated in linear constraint databases by piecewise linear function. In this chapter we will discuss the query evaluation in linear constraint database based on piecewise linear approximation, which is a faster approximation that preserves high precision and recall. The presentation below is based on our work in [6].

Suppose that there are a sequence of time series weather data $S = \{(d_1, ht_1, lt_1),\ (d_2, ht_2, lt_2),\ \cdots,\ (d_n, ht_n, lt_n)\}$ where d_is $(1 \leq i \leq n)$ are consecutive days, ht_is $(1 \leq i \leq n)$ are daily high temperatures and lt_is $(1 \leq i \leq n)$ are daily low temperatures. The data are saved in relational database $R(day, ht, lt)$. The following SQL query finds the pair of days such that the first day's low temperature is higher than the second day's high temperature, which is shown in the following query 6.1:

$$\begin{array}{ll} \textbf{Select} & R1.day,\ R2.day \\ \textbf{from} & R\ R1,\ R\ R2 \\ \textbf{where} & R1.lt > R2.ht \end{array} \tag{6.1}$$

In linear constraint database system (such as MLPQ), we can use piecewise linear approximation to approximate the hts and lts in the relation R. Suppose that we use a relation $A(day, ht, lt)$ in MLPQ which has k linear constraint tuples to approximate R such that at each day d_i, the max error for the temperature is less than some constant error tolerance Ψ. The constraint tuples are as following:

Day	**ht**	**lt**	
d	ht	lt	$d_1 \leq d < dd_2,\quad ht = h_1 + s_1 * (d - d_1),\quad lt = l_1 + t_1 * (d - d_1)$
d	ht	lt	$dd_2 \leq d < dd_3,\quad ht = h_2 + s_2 * (d - dd_2),\quad lt = l_2 + t_2 * (d - dd_2)$
...	...	...	...
d	ht	lt	$dd_k \leq d < d_n,\quad ht = h_k + s_k * (d - dd_k),\quad lt = l_k + t_k * (d - dd_k)$

Table 6.1: The constraint tuples in linear constraint database

Then in constraint database system, the previous query can be written as the following SQL query 6.2:

$$\begin{array}{ll} \textbf{Select} & A1.day,\ A2.day \\ \textbf{from} & A\ A1,\ A\ A2 \\ \textbf{where} & A1.lt > A2.ht \end{array} \tag{6.2}$$

The constraint database system (such as MLPQ) can evaluate the above query and give linear constraint database output. From the linear constraint database output, we can create the pair of days output by the export conversion described in Chapter 4.

Suppose that the set of tuples returned by query 6.1 is R, the set of tuples as the export conversion result from query 6.2 is C. We can consider the approximate evaluation of query 6.2 as an information retrieval system of query 6.1. Then in this system, each tuple t of possible pair of days must fall into one of the following four categories, as shown in Figure 6.1:

1. $t \in R$ and $t \in C$, we call this as "Relevant Retrieved"
2. $t \in R$ and $t \notin C$, we call this as "Relevant Not Retrieved"
3. $t \notin R$ and $t \in C$, we call this as "Non-Relevant Retrieved"
4. $t \notin R$ and $t \notin C$, we call this as "Non-Relevant Not Retrieved"

From [17], the two major measures are:

$$Precision = \frac{Number\_Retrieved\_Revelent}{Number\_Total\_Retrieved}$$

$$Recall = \frac{Number\_Retrieved\_Revelent}{Number\_Possible\_Revelent}$$

In our example the $Number\_Total\_Retrieved$ is $|C|$, the $Number\_Possible\_Revelent$ is $|R|$.

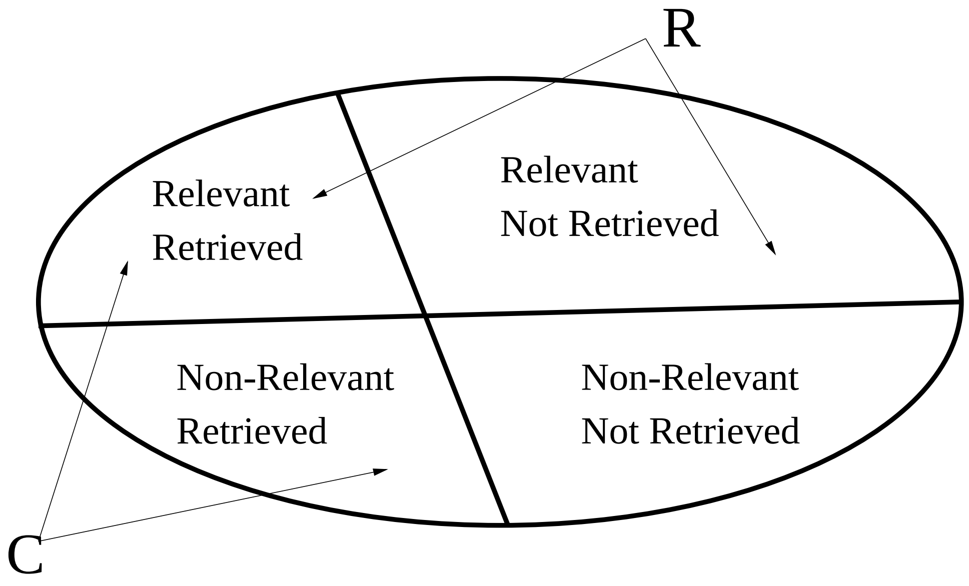


Figure 6.1: Search Result

6.1 Experimental Results

We did some experiments using the temporal dataset containing 10 years' daily high temperature, daily low temperature datasets between the year 1987 and 1996 from the weather station in Nebraska State (station number: 252820). We use different Ψ values to restrict the piecewise linear transformation. The original weather data comes from the website of the National Climatic Data Center at http://www.ncdc.noaa.gov.

We used the running window method to smooth the original data. The running window size is set to 7.

$R_{high}(day,\ high\_temp)$ is the high temperature relation, $R_{low}(day,\ low\_temp)$

Ψ	**#Pieces in High Temperature Dataset**	**# Pieces in Low Temperature Dataset**
—	3,653	3,653
1.0	1,426	1,084
2.0	790	594
4.0	428	335
2.0	197	140

Table 6.2: Number of Pieces in Datasets

is the low temperature relation. During the comparison, R_{high} and R_{low} are renamed as R'_{high} and R'_{low} respectively. Also, we get the relation $R(day,\ high\_temp,\ low\_temp)$ by join the relation R_{high} and R_{low}.

The following are some experimental tests for simple and composite algebraic queries based on these three relations. In these tests, we use MLPQ system to evaluate the constraint queries and after getting the constraint tuple output, use export conversion as in Chapter 4 to get the pair of days.

Simple Algebraic Queries

Example 6.1.1 We try to find all pair of days such that for each the high temperature in one day is greater than or equal to that in the other.

The SQL query is as follows.

```
select R_1.day, R_2.day
from R_high R_1, R_high R_2
```

```
where R_1.high_temp >= R_2.high_temp;
```

The Datalog query in MLPQ is as follows.

```
Pair(day1, day2) :- R_high(day1, high_temp1),
                    R_high(day2, high_temp2),
                    high_temp1 >= high_temp2.
```

The test results are as Table 6.3.

Ψ	**MLPQ Constraints**	**Actual # Solutions**	**Precision**	**Recall**
1.0	2,006,001	6,645,646	99.39%	99.49%
2.0	1,208,010	6,645,646	98.54%	98.67%
4.0	235,639	6,645,646	96.83%	96.97%
8.0	51,681	6,645,646	93.39%	93.53%

Table 6.3: $R_{high} >= R'_{high}$

Example 6.1.2 We try to find all pair of days such that for each the low temperature in one day is greater than or equal to that in the other.

The SQL query is as follows.

```
select R_1.day, R_2.day
from R_low R_1, R_low R_2
where R_1.low_temp >= R_2.low_temp;
```

The Datalog query in MLPQ is as follows.

```
Pair(day1, day2) :- R_low(day1, low_temp1),
                    R_low(day2, low_temp2),
                    low_temp1 >= low_temp2.
```

The test results are as Table 6.4.

Ψ	**MLPQ Constraints**	**Actual # Solutions**	**Precision**	**Recall**
1.0	2,006,343	6,644,379	99.29%	99.42%
2.0	697,901	6,644,379	98.35%	98.51%
4.0	235,624	6,644,379	96.38%	96.51%
8.0	51,676	6,644,379	92.84%	92.99%

Table 6.4: $R_{low} >= R'_{low}$

Composite Algebraic Queries:

Example 6.1.3 We try to find all pair of days such that for each the high temperature in one day is greater than or equal to that in the other and the low temperature in one day is also greater than or equal to that in the other.

The SQL query is as follows.

```
select R_1.day, R_2.day
from R R_1, R R_2
```

```
where R_1.high_temp >= R_2.high_temp
  and R_1.low_temp >= R_2.low_temp;
```

The Datalog query in MLPQ is as follows.

```
Pair(day1, day2) :- R(day1, high_temp1, low_temp1),
                    R(day2, high_temp2, low_temp2),
                    high_temp1 >= high_temp2,
                    low_temp1 >= low_temp2.
```

The test results are as Table 6.5.

Ψ	**MLPQ Constraints**	**Actual # Solutions**	**Precision**	**Recall**
1.0	1,836,631	6,091,441	99.30%	99.44%
2.0	639,797	6,091,441	98.40%	98.66%
4.0	215,649	6,091,441	96.45%	96.67%
8.0	46,449	6,091,441	92.72%	92.68%

Table 6.5: $R_{high} >= R'_{high}$ **and** $R_{low} >= R'_{low}$

Example 6.1.4 We try to find all pair of days such that for each the high temperature in one day is greater than or equal to that in the other or the low temperature in one day is also greater than or equal to that in the other.

The SQL query is as follows.

```
select R_1.day, R_2.day
from R R_1, R R_2
where R_1.high_temp >= R_2.high_temp
   or R_1.low_temp >= R_2.low_temp;
```

The Datalog query in MLPQ is as follows.

```
Pair(day1, day2) :- R(day1, high_temp1, low_temp1),
                    R(day2, high_temp2, low_temp2),
                    high_temp1 >= high_temp2.
Pair(day1, day2) :- R(day1, high_temp1, low_temp1),
                    R(day2, high_temp2, low_temp2),
                    low_temp1 >= low_temp2.
```

The test results are as Table 6.6.

Ψ	**MLPQ Constraints**	**Actual # Solutions**	**Precision**	**Recall**
1.0	2,175,713	7,198,584	99.39%	99.48%
2.0	755,959	7,198,584	98.56%	98.60%
4.0	255,614	7,198,584	96.93%	97.00%
8.0	56,908	7,198,584	94.05%	94.37%

Table 6.6: $R_{high} >= R'_{high}$ **or** $R_{low} >= R'_{low}$

From above we can see that the approximate evaluation gives much fewer number of output tuples because of the data compression in piecewise linear approximation. It also preserves high precision and recall. Our work shows that there is a trade-off between space and accuracy for the evaluation (which is depend on the approximation error tolerance). For different precision and recall requirements we can choose different approximation error tolerance such that it satisfies the precision and recall requirements in query evaluation and use as less space as possible.

Chapter 7

Query Optimization

In this chapter, we propose a new query optimization strategy based on the hypergraph representation of queries. The presentation below is based on our work in [19, 22].

Since the introduction of Wong and Youssefi's algorithm in 1976, there were many important discoveries on the efficient partitioning of hypergraphs [2, 15, 22]. To our knowledge, these more recent discoveries were not used yet to develop new query optimization strategies for relational algebra queries. In this chapter, we describe a new optimization strategy that requires smaller intermediate size relations in the evaluation. Also, our optimization strategy tends to yield expressions that can be evaluated easily by parallel processors.

Hypergraph Partitioning: We are considering in this chapter hyperedge partitioning of hypergraphs. In such a partitioning each hyperedge belongs to exactly one of two parts. However, a vertex may belong to several parts. We call a hyperedge a *cross edge* if it contains vertices that belong to both parts.

The rest of this section is structured as follows. Section 7.1 discusses hypergraph partitioning-based query optimization on natural join queries. Section 7.2 compares Wong-Youssefi's optimization algorithm with hypergraph partitioning-based optimization algorithm on an example query. Section 7.3 discuss the optimization of queries with select operation. Section 7.4 discuss the optimization for general queries. Section 7.5 briefly discuss the hypergraph partition heuristics, which can be used for query optimization.

7.1 Hypergraph Partitioning-based Optimization for Join Operations

Let us first explore the case when there are only natural join operations in the query. To optimize such a query, we first represent it as a hypergraph. Then we find a hyperedge partitioning of the hypergraph. After the partition, we evaluate each part separately and finally join them.

For large hypergraphs, each part is recursively partitioned into smaller parts until each part contains less than four relations. This recursive algorithm yields an evaluation tree as shown in Figure 7.1.

Now we will describe the hypergraph partitioning-based optimization strategy.

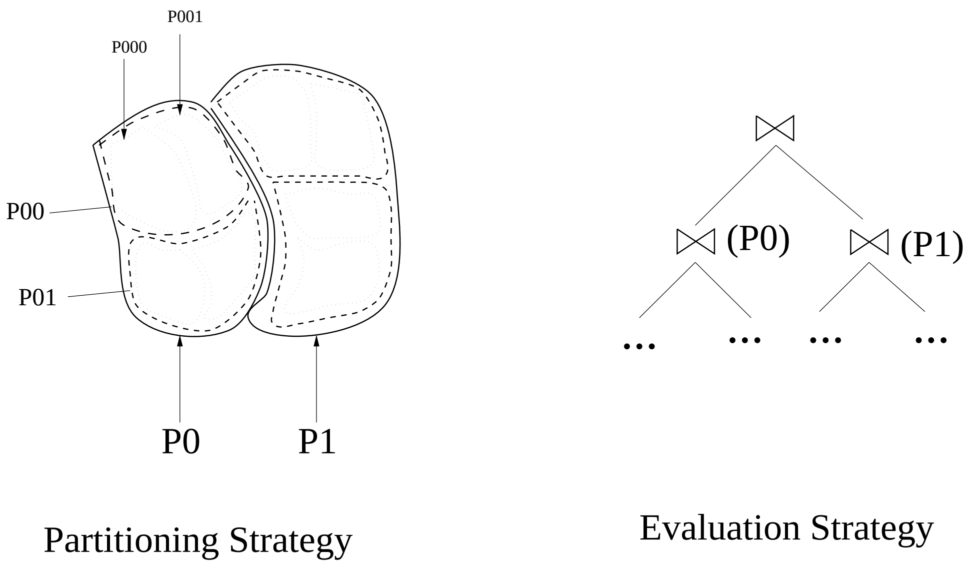


Figure 7.1: Recursively Partitioning-based Optimization

HYPERGRAPH PARTITIONING-BASED OPTIMIZATION ALGORITHM:

Input: A relational algebra query with only natural operations

Output: A binary tree which gives an evaluation strategy.

Local Vars: $root$ is a binary tree node type variable which has $left$ and $right$ pointers and $item$ as a hypergraph/subgraph. s is a stack which has the same element type as $root$. a, p_1 and p_2 are tree node type variables.

```
        Represent the query using a hypergraph H=(V,E).
        root− >item = H; root− >left = NULL; root− >right=NULL;
        push(s, root);
        while stack $s$ is not empty do
begin
  a = pop(s);
  if a− >item has more than one relation, then
    begin
      partition the hypergraph a− >item into 2 parts $p_1$ and $p_2$
      create two new tree nodes $t_1$ and $t_2$
      $t_1$− >item = $p_1$;
      $t_1$− >left=NULL;
      $t_1$− >right=NULL;
      $t_2$− >item = $p_2$;
      $t_2$− >left=NULL;
      $t_2$− >right=NULL;
      a− >left=$t_1$;
      a− >right=$t_2$;
      push(s, $t_1$);
      push(s, $t_2$);
    end
```

```
**end**
```

Partitioning Heuristic: Suppose that we partition a hypergraph into two parts $P0$ and $P1$. The more common attributes (vertices shared by two or more hyperedges) there are within the relations of $P0$, the smaller $P0$ will be in general. Similarly, the more common attributes $P1$ contains, the smaller it will be in general. Since we want to have small intermediate size relations, our goal is to find a partition such that (1) both parts have about the same number of hyperedges and (2) the number of common attributes within each part is maximal.

Clearly, the whole graph contains a fixed number of common attributes. Most of these occur also as common attributes in $P0$ or in $P1$. However, there are some common attributes which are shared only by a hyperedge in $P0$ and a hyperedge in $P1$. It is the number of these common attributes that we want to minimize.

As an approximation of this, we may choose a partition that minimizes the number of hyperedges that cross the two parts. This is called the *minimum partitioning* problem for hypergraphs for which several good heuristics have been already developed that we can use (see Section 7.5).

Extended strategy with Semi-joins: Even though we minimized the number of cross edges during the partition, we could still take some advantage of the common attributes found in the cross edges, by semi-joining them with the adjacent edges in the other part. This method would alleviate the fact that there are still some attributes that are common only between the two hyperedges that belong to different parts. Note however, that even under this assumption, it still makes sense to choose a minimal partitioning of the hypergraph, in this case not for maximizing the number of common attributes in the two parts, but for minimizing the number of semi-join operations that need to be introduced in each partition step.

Next let's use an example to illustrate the strategy.

Example 7.1.1 Suppose that there are 6 relations, $A(a_1, a_2, a_3)$, $B(a_3, a_4, a_5)$, $C(a_5, a_6, a_7)$, $D(a_7, a_8, a_9)$, $E(a_9, a_{10}, a_{11})$, $F(a_{11}, a_{12}, a_1)$. We need to compute the natural join of the 6 relations, that is, to calculate:

$$A \bowtie B \bowtie C \bowtie D \bowtie E \bowtie F \tag{7.1}$$

The algorithm first construct the hypergraph corresponding to the query 7.1 as in Figure 7.2.

The recursively partitioning process is shown in Figure 7.3, which yields the evaluation tree as shown in Figure 7.4.

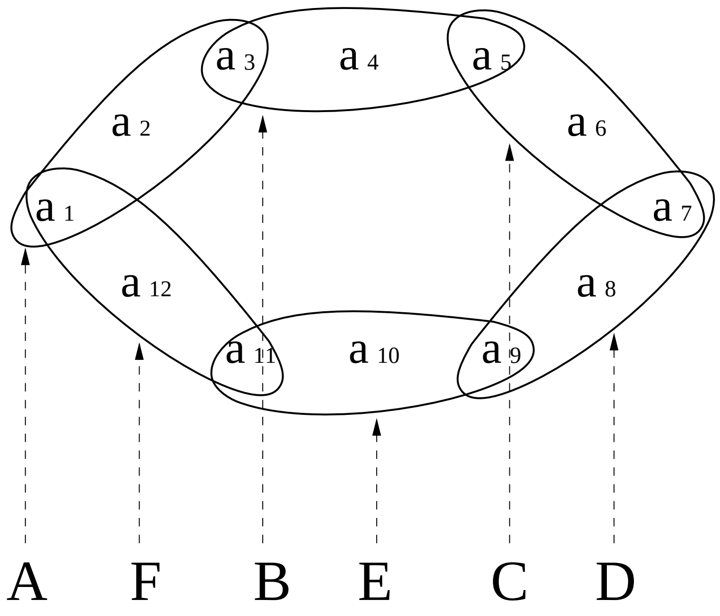


Figure 7.2: Hypergraph for A Natural Join Query

The evaluation is bottom-up on the binary tree. The following is the algorithm for creating the evaluating strategy from the binary tree created in hypergraph-partitioning based optimization algorithm.

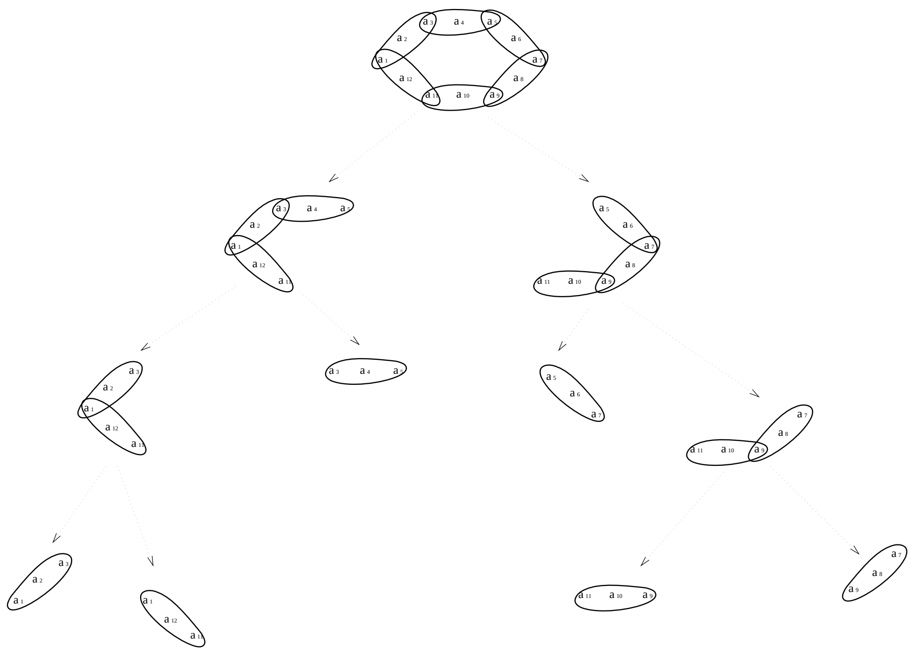
EVALUATION STRATEGY ALGORITHM:

Name Evaluation strategy

Input: A binary tree created by hypergraph-partitioning based optimization algorithm.

root is the root of the tree.

Output: An evaluation strategy.



a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8 a_9 a_{10} a_{11} a_{12}

Figure 7.3: Hypergraph Partitioning Solution

Local Vars:le and *re* are evaluation results.

```
        **if** ((root$-$ >left==NULL) && (root$-$ >right==NULL))
return(root$-$ >item);
        **else**
**begin**
   le=evaluate(root$-$ >left);
   re=evaluate(root$-$ >right);
   return(le $\bowtie$ re);
**end**
```

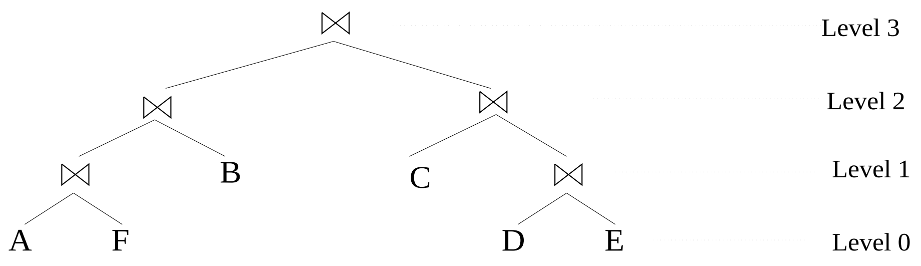


Figure 7.4: Evaluation Tree

Using the above algorithm, we get the following relational algebra expression for the query:

$$((A \bowtie B) \bowtie F) \bowtie ((C \bowtie D) \bowtie E)$$

7.2 Comparison with Wong-Youssefi's Optimization Algorithm

In this section, we compare our optimization result to the result of Wong-Youssefi's optimization algorithm.

Wong-Youssefi's optimization strategy first tries to find and reduce a small hyperedge or a hyperedge that when taken away would disconnect the hypergraph. Note that in Figure 7.2, there is neither a small hyperedge nor a hyperedge that when taken away would disconnect the hypergraph. Hence, we have to arbitrarily choose one hyperedge to reduce the hypergraph. Suppose we arbitrarily choose hyperedge A. Then we have:

$B = B \propto A, \quad F = F \propto A.$

From Wong-Youssefi's algorithm, now both B and F become "small" relations. Next we can reduce either B or F. We arbitrarily choose B to reduce, we now have:

$C = C \propto B.$

Next we can reduce D, then E and then F, the final evaluation result is:

$$
\begin{aligned}
&B = B \propto A\\
&F = F \propto A\\
&C = C \propto B\\
&D = D \propto C\\
&E = E \propto D\\
&F = F \propto E\\
&H_1 = F \bowtie E\\
&H_2 = D \bowtie H_1\\
&H_3 = C \bowtie H_2\\
&H_4 = B \bowtie H_3\\
&R = A \bowtie H_4
\end{aligned}
$$

The evaluation tree for Wong-Youssefi algorithm is shown in Figure 7.5 (in this figure we only show the $\bowtie$ operations and do not show the $\propto$ operations):

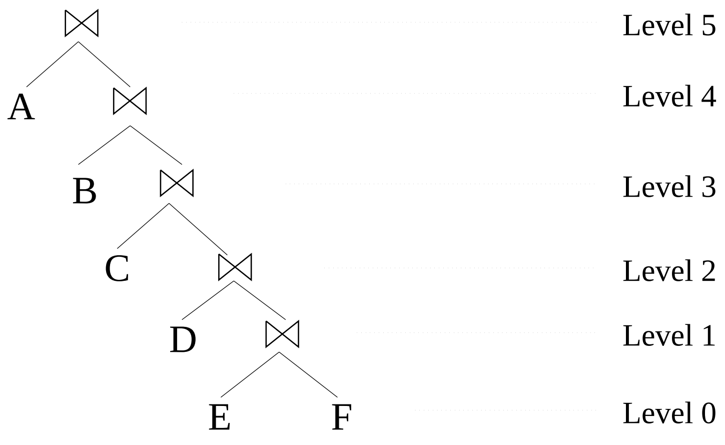


Figure 7.5: Wong-Youssefi's Solution

7.2.1 Comparison of Costs and Parallel Evaluation

In order to estimate the output cost for joins in each level of the evaluation tree, let us suppose that each relation has 1000 tuples, that is, $T_A = 1000, ..., T_J = 1000$, and suppose that for each attribute a_i in each relation the different number of attribute values is 100, that is, $I_{a_i} = 100$ for each $1 \leq i \leq 12$. We also assume that there are no dangling tuples, hence semi-join operations do not shrink the size of the relations. From the formula in [32], the number of tuples in the natural join of the two relations $R(B_1, \cdots, B_k, A_1, \cdots, A_j)$ and $S(B_1, \cdots, B_k, C_1, \cdots, C_m)$ is:

$$\frac{T_R T_S}{I_{B_1} \cdots I_{B_k}}$$

Based on the above assumption and the formula, we can estimate the number of tuples of the intermediate relation in each level of the evaluation tree. The estimates for Wong-Youssefi's algorithm is shown in Figure 7.6, while the estimates for our algorithm is shown in Figure 7.7. It can be seen that the maximum size of any intermediate relation in Wong-Youssefi's algorithm is 10^7, while for our new algorithm it is only 10^5. Therefore, our optimization algorithm can be expected to perform faster in this example.

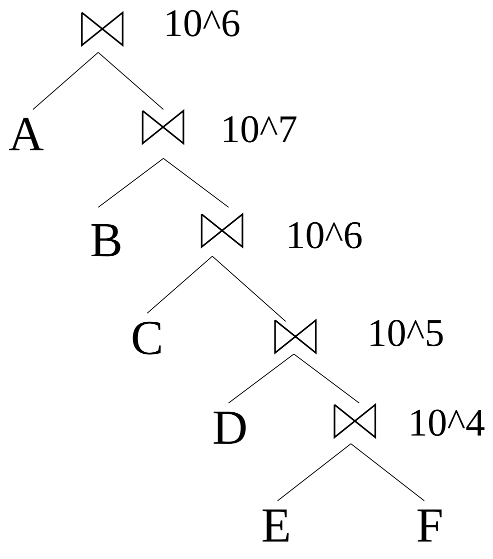


Figure 7.6: Number of Tuples for Wong-Youssefi's Algorithm

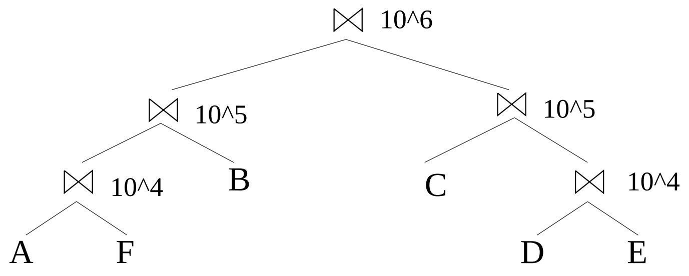


Figure 7.7: Number of Tuples for New Algorithm

Parallel Evaluation: As we can see in Figure 7.6 the structure of the query evaluation tree of Wong-Youssefi's algorithm tends to be a long chain unless there are disconnecting edges which are chosen frequently. In contrast, as Figure 7.7 suggests, our algorithm is returning well-balanced query optimization trees. Therefore, the output of our algorithm is much more easily parallelized. It also should be pointed out that the partitioning itself can be also done by parallel computers (see Section 7.5).

7.3 Hypergraph Partitioning-based Optimization of Queries with Joins and Select Conditions

Now we discuss a little bit more complex case. We discuss the join operations with selection conditions. That is, the case of expressions with form:

$$R = \sigma_{F_1 \wedge \cdots \wedge F_k}(R_0 \times R_1 \times \cdots \times R_n)$$

.

Here each condition F_i is any condition expression except the logic **AND** of two expressions.

The query with join operations and select conditions can also be represented as a hypergraph. In this case, we represent both the relations and the select conditions with hyperedges.

In order create a hypergraph to represent the query, we first create vertices for the hypergraph. Generally, each attribute in a relation will create a vertex, however, if there is an equal condition between two attributes of two relations, which means that there is a natural join between these two relations, then these two vertices should be merged into one.

After creating the vertices, we create hyperedges for both relations and select conditions. Each relation is represented as a hyperedge which contains all vertices corresponding to the attributes in this relation. For a select condition F_i, if it is not a equal condition of two attributes (which has been represented by merging the two equal vertices) and it is related to attributes $a_1, a_2, \cdots, a_n$, then it creates a hyperedge E_{F_i} that include vertices $\{a_1, a_2, \cdots, a_n\}$. The following algorithm illustrates this.

HYPERGRAPH CREATING ALGORITHM:

Input: A relational algebra query with join operations and select conditions.

Output: A hypergraph H=(V,E) which is corresponding to the query.

```
Local Vars: $e_1$ and $e_2$ are variables for hyperedge
    V=Φ;
  E=Φ;

  for each relation $r_i$ in the query
    for each attribute $a_j$ in relation $r_i$
      V=V ∪ {$a_j$}
  for each equal select condition of form $a_j = a_k$
      V=V − {$a_j$} /* merge $a_j$ and $a_k$ */

  for each relation $r_i$ in the query
    begin
      $e_1$=Φ;
      for each attribute $a_j$ in relation $r_i$
      if $a_j$ has been merged with another vertex $a_k$, then
        $e_1 = e_1 \cup \{a_k\}$
      else
        $e_1 = e_1 \cup \{a_j\}$
        assign the edge weight of $e_1$ be MAXINT
      E=E ∪ {$e_1$}
    end
```

```
for each select condition $c_i$ in the query
  if it is not the equal of two attributes, then
    begin
      $e_2$=$\Phi$;
      for each attribute $a_j$ in condition $c_i$
        if $a_j$ has been merged with another vertex $a_k$, then
          $e_2 = e_2 \cup \{a_k\}$
        else
          $e_2 = e_2 \cup \{a_j\}$
        assign the edge weight of $e_2$ be 1
      E=E $\cup$ $\{e_2\}$
  end
```

In the following example, we use solid line hyperedges to represent relations, and dashed line hyperedges to represent conditions, as in [32].

Example 7.3.1 Suppose that there are 5 relations $R_0(A, B)$, $R_1(C, D)$,

$R_2(E, F)$, $R_3(G, H, I)$, $R_4(J, K)$. For the query:

$$\sigma_{A=E \wedge C=F \wedge D<G \wedge H=J \wedge I<K}(R_0 \times R_1 \times R_2 \times R_3 \times R_4) \tag{2}$$

The hypergraph for this query is shown in Figure 7.8.

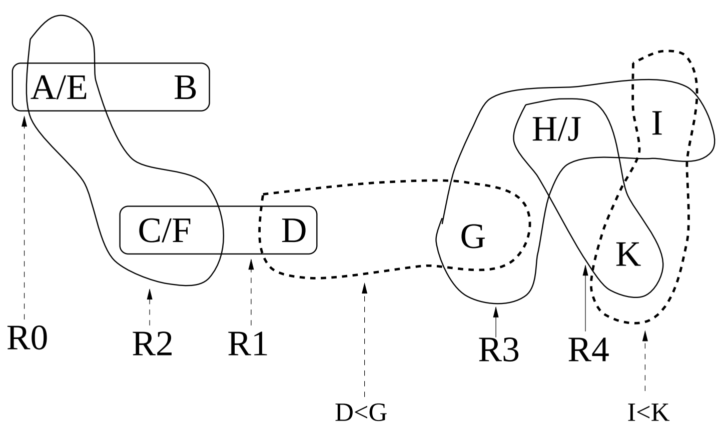


Figure 7.8: Hypergraph for SQL Query (2)

In Figure 7.8, the equal conditions $A = E$, $C = F$ and $H = J$ can be expressed by merging vertices (A, E), (C, F) and (H, J). Conditions $D < G$ and $I < K$ can be expressed by two hyperedges (see the dashed line hyperedges in Figure 7.8).

Since generally the evaluation of non-equal conditions is more costly than equal joins and creates more tuples. Equal joins should be evaluated before evaluating non-equal conditions. For a hypergraph, we should first evaluate each part that is connected by equal joins. Then we evaluate the hyperedges of non-equal relations. For each part that is connected by equal join hyperedges, it is evaluated just as in Section 7.1. In the previous algorithm, by assigning different edge weight for hyperedges (MAXINT for equal condition, 1 for other cases), the minimal-cut hyperedge partitioning algorithm will try to partition the hypergraph into two parts that is connected by non-equal conditions, which means that the equal conditions will be evaluated earlier. For example, Figure 7.8 should be evaluated as:

$$P_0 = \sigma_{A=E \wedge C=F}(R_0 \times R_1 \times R_2)$$

$$P_1 = \sigma_{H=J \wedge I<K}(R_3 \times R_4)$$

$$R = \sigma_{D<G}(P_0 \times P_1)$$

Small Relations

A relation with select condition of the form $A = a$ where A is an attribute and a is a constant is said to be a "small" relation, because generally the condition $A = a$ will greatly eliminate the number of tuples.

If a relation joins to a "small" relation, it is likely that the result is also not much large. Hence this non-equal join operation should be done earlier than other non-equal joins. The following example demonstrates this idea.

We can change the above hypergraph creating algorithm to deal with small relations. After we create the hypergraph,

Example 7.3.2 Suppose that there are 7 relations $R_0(A, B)$, $R_1(C, D)$, $R_2(E, F)$, $R_3(G, H)$, $R_4(U, V)$, $R_5(W, X)$, $R_4(Y, Z)$. For the query:

$$R = \sigma_{A=E \wedge C=F \wedge D<G \wedge H>U \wedge U=2 \wedge V=W \wedge X=Y}(R_0 \times R_1 \times R_2 \times R_3 \times R_4 \times R_5 \times R_6)$$

(3)

The hypergraph for this query is shown in Figure 7.9.

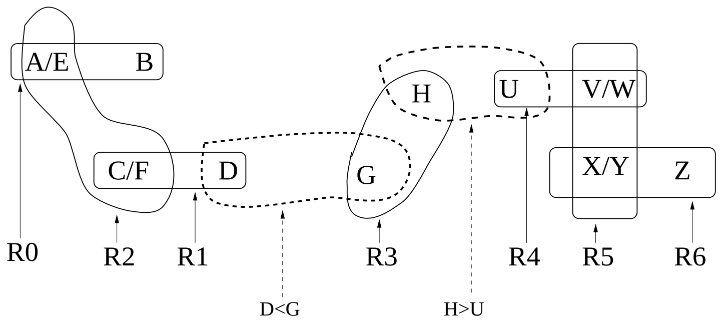


Figure 7.9: Hypergraph for SQL Query (3)

In this example, because we have the select condition $U = 2$, the relation R_4 is a small relation. And because R_5 and R_6 have equal join relations to small relation R_4, they all from a small relation subpart of the graph. In Figure 7.10, there are two non-equal joins ($D < G$ and $H > U$), because the non-equal join $H > U$ is a join to a small relation, it should be evaluated earlier than join $D < G$.

The evaluation strategy should be:

$$P_{00} = \sigma_{A=E}(R_0 \times R_2)$$

$$P_0 = \sigma_{C=F}(P_{00} \times R_1)$$

$$P_{100} = \sigma_{U=2} R_4$$

$$P_{10} = \sigma_{V=W}(P_{100} \times R_5)$$

$$R_{small} = \sigma_{X=Y}(P_{10} \times R_6)$$

$$P_1 = \sigma_{H>U}(R_3 \times R_{small})$$

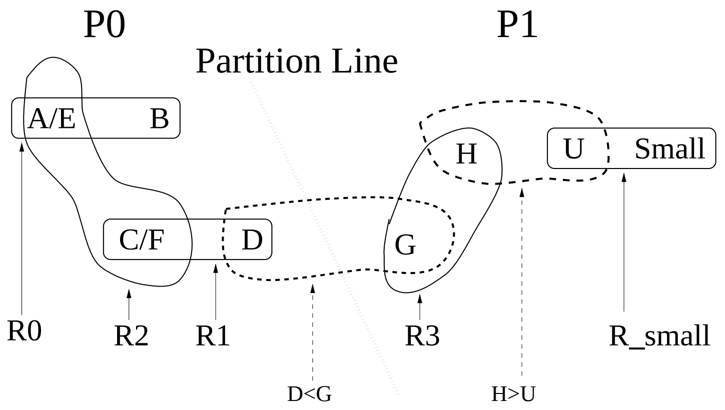


Figure 7.10: Small Relations in Hypergraphs

$$R = \sigma_{D<G}(P_0 \times P_1)$$

7.4 Hypergraph Partitioning-based Optimization of General Queries

Now we discuss the optimization for general query expression. I.e., the query with the form:

$$R = \Pi_{a_1,\cdots,a_m}\sigma_{F_1\wedge\cdots\wedge F_n}(R_1 \times \cdots \times R_k).$$

Where $a1$, $\cdots$, a_m are attributes of relations, F_1 $\cdots$, F_n are conditions. The R_1, $\cdots$, R_k are any relations.

The hypergraph for this query is the same as the quey without the projection operations. The idea for optimizing these queries is to push the projections as far down the evaluation tree as possible, we can use the technology described in

Chapter 11 of [32] within our partitioning-based optimization to handle these cases. Concretely, in the partitioning-based optimization, after we compute the join result of relations, we use the technology in [32] to do projection operations to eliminate unnecessary attributes as early as possible to get efficient optimization solution.

7.5 Hypergraph Partitioning Heuristics

In previous sections, we propose a new hypergraph partitioning based query optimization algorithm. Now we will briefly discuss the classes of hypergraph partitioning heuristics.

Basically, hypergraph partitioning algorithms try to partition a hypergraph into a number of parts, such that the size of each partition satisfies some partition size constraints, and the number of hypergraphs across the partitions is minimized. The hypergraph partitioning problem is NP-hard.

There are a lot of hypergraph partitioning algorithms. The earliest sets of partitioning algorithms are *iterative refinement partitioning algorithms*. In these algorithms, an initial partitioning is computed (often obtained randomly) and then the partitioning is refined by repeatedly moving vertices between subsets of the partition to reduce the hyperedge-cut. Kernighan-Lin (KL) heuristic and the faster Fiducia-Matteyses (FM) refinement heuristic [9] are two examples for this class of heuristics. The partition produced by these methods are usually poor especially for large hypergraphs.

In 1990s, a new class of partitioning algorithms was developed that are based on the multi-level paradigm. Following this paradigm, a sequence of successively smaller graphs are constructed, usually based on matching algorithms for graphs or hypergraphs. A bisection of the smallest graph is computed, and the bisection is then successively projected to the next less coarsened hypergraph, and at each level an iterative refinement algorithm such as FM is used to further improve the bisection, as illustrated in Figure 7.11.

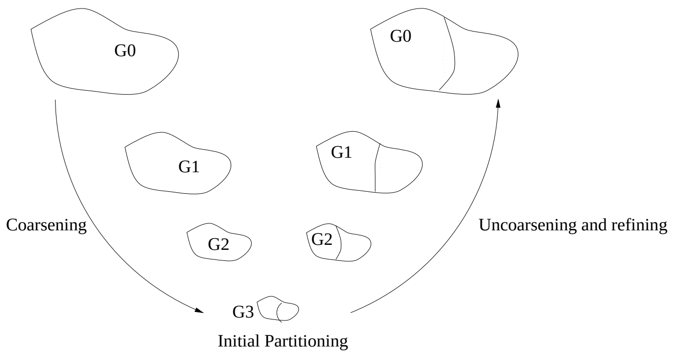


Figure 7.11: Multilevel Paradigm for Hypergraph Partitioning

In [22], we proposed a new algorithm, which is called "Multilevel Cooperative Search Algorithm" for hypergraph partitioning. In the new partitioning algorithm, hypergraph is coarsened to form a successively smaller sequence of hypergraphs, and during the partitioning process, each level of hypergraph cooperatives by sharing information between adjacent levels. The new partitioning algorithm is able to give better partitioning results than state-of-art multilevel hypergraph partitioners. Figure 7.12 shows the paradigm for the new partitioning algorithm. This can be used to find better query optimization solutions when using the hypergraph partitioning based optimization as described in previous sections.

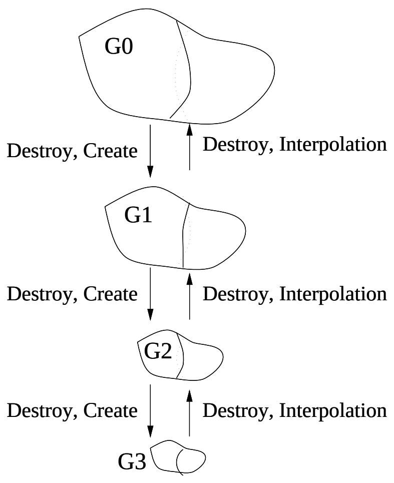


Figure 7.12: Multilevel Cooperative Search Paradigm

Chapter 8

Conclusion

In Chapter 3 we presented a fast value-by-area cartogram transformation algorithm that can be used as a subroutine within animation algorithm. We also presented the idea of serial and hybrid animation methods that further enhance the speed of the animation while avoiding excessive cell shape distortion accumulation. The speed of our animation methods avoids the need to pre-compute all the snapshots of the animation. The system can animate databases that contain several thousands or even more snapshots, which is impractical for others that using pre-computation strategy.

Chapter 4 described the piecewise linear approximation algorithms, which can be used to interpolate and compress data. Chapter 5 showed how to combine the work in Chapter 4 and Chapter 5 to get more efficient cartogram animation.

In Chapter 6, we dealt with approximate evaluation of queries. Approximate evaluation runs very fast and can get high precision and recall. it is extremely suitable for large volume GIS spatiotemporal databases.

In Chapter 7, we proposed a new query optimization algorithm. Our algorithm uses new discoveries on the efficient partitioning of hypergraphs to create more efficient and parallelable query evaluation strategies.

8.1 Open Problems and Future Work

The animation system can be easily extended to display different levels of animation. For example, when animating the population distribution, at top level, it displays the value-by-area cartogram with each country as a cell. At this level, when we want to see the population detail of the United States, we can click on the cell of the United States, then it begins to animate the United States population cartogram with each state as a cell. Then, if we want to further study the population of the Nebraska state, we can click on the cell of Nebraska and see the population distribution at county level, and so on.

Our computer experiments show that our cartogram algorithm works much faster than traditional cartogram algorithms [7, 11, 13, 31]. However, in general, it is difficult to analyze theoretically the computational complexity of value-by-area cartogram algorithms, and therefore no such analysis is contained in the papers [7, 11, 13, 31]. For future work, we are planing to do some theoretical analysis for the complexity of the algorithm.

Another work is to study and compare the quality of the cartogram algorithms. Our algorithm gives better looking cartograms. We are planning to study the quality of cartogram based on similarity measure between the cartogram and the original geographic map to see how to improve the quality of the cartogram animation.

For piecewise linear approximation, an open problem is to find an optimal piewise linear approximation agorithm which has $O(N)$ complexity.

In Chapter 7, we propose a new hypergraph partitioning based query optimization strategy that requires less space for the intermediate relations than the optimized expressions given by earlier algorithms. For future work, we can try to combine the hypergraph partitioning based query optimization with cost-estimations on physical structure of database such as the existence of indices, the number of tuples in each database. This may give more precise estimations for the operations and thus yield more efficient evaluation strategies.

Bibliography

[1] P. K. Agarwal and K. R. Varadarajan. Efficient Algorithms for Approximating Polygonal Chains. *Journal of Discrete & Computational Geometry*, 23:273–291, 2000.

[2] C.J. Alpert and A.B. Kahng. Recent Directions in Netlist Partitioning: a Survey. *Integration, the VLSI Journal*, 19:1–18, 1995.

[3] A. Brodsky, V. Segal, J. Chen, and P. Exarkhopoulo. The CCUBE Constraint Object-Oriented Database System. In Proc. ACM SIGMOD, pages 577–579, 1999.

[4] R. Chen, M. Ouyang, and P. Revesz. Approximating Data in Constraint Databases. In *Proc. Symposium on Abstraction, Reformulation and Approximation, Horse shoeBay, Texas*, volume 1864 of *Lecture Notes in Artificial Intelligence*, pages 124–143. Springer-Verlag, July 2000.

[5] R. Chen, M. Ouyang, and P. Revesz. A Time Series Data Model. In *2001 Annual Meeting, Association of American Geographers*, New York, NY, USA,

2001.

[6] R. Chen, M. Ouyang, and P. Z. Revesz. Approximate SQL Query Evaluation in Linear Constraint Databases. *Constraints*, (submitted).

[7] J. A. Dougenik, N. R. Chrisman, and D. R. Niemeyer. An Algorithm to Construct Continuous Area Cartograms. *Professional Geographer*, 37(1):75–81, 1985.

[8] David H. Douglas and Thomas K. Peucker. "algorithms for the reduction of the number of points required to represent a digitized line or its caricature". *Canadian Cartographer*, 10(2):112–122, 1973.

[9] C. Fiduccia and Matteyses. A Linear Time Heuristics for Improving Network Partitions. In *Proc. 19th ACM/IEEE Design Automation Conference*, pages 175–181. IEEE Computer Society Press, 1982.

[10] S. Grumbach, P. Rigaux, and L. Segoufin. The DEDALE System for Complex Spatial Queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 213–224, Seattle, Washington, USA, 1998.

[11] S. M. Gusein-Zade and V. S. Tikunov. A New Technique for Constructing Continuous Cartograms. *Geography and Geographic Information Systems*, 20(3):167–173, 1993.

[12] S. L. Hakimi and E. F. Schmeichel. Fitting polygonal functions to a set of points in the plane. In *CVGIP: Graph. Mod. Image Proc.*, pages 132–136, 1991.

[13] D. H. House and C. J. Kocmoud. Continuos Cartogram Construction. In *Proc. of IEEE Visualization Conference*. IEEE, 1998.

[14] P. Kanjamala, P.Z. Revesz, and Y. Wang. MLPQ/GIS: A GIS using linear constraint databases. In C. S. R. Prabhu, editor, *Proceedings of the 9th COMAD International Conference on Management of Data*, pages 389–393. Tata McGraw Hill, 1998.

[15] G. Karypis and V. Kumar. Muitilevel k-way Hypergraph Partitioning. In *Proc. 36th ACM/IEEE Design Automation Conference*. Association for Computing Machinery, 1999.

[16] C. J. Kocmoud and D. H. House. Cartogram Animation of U.S. Population Cartograms from 1900 to 1996. In *www-viz.tamu.edu/faculty/house/cartograms/DecadeAnim.html*.

[17] G. Kowalski. *"Information Retrieval Systems – Theory and Implementation"*. Kluwer Academic Publishers, 1997.

[18] P. O'Neil. *Database – Principles Programming Performance*. Morgan Kaufmann Publishers Inc., 1994.

[19] M. Ouyang and P. Z. Revesz. A Hypergraph Partitioning-based Query Optimization Strategy (manuscript). 2000.

[20] M. Ouyang and P. Z. Revesz. Algorithms for Cartogram Animation. In *Proc. 4th International Database Engineering and Applications Symposium*, pages 231–235, 2000.

[21] M. Ouyang and P. Z. Revesz. Animations for Spatiotemporal Databases. *Geoinformatica*, (submitted).

[22] M. Ouyang, M. Toulouse, and et. al. Multilevel Cooperative Search: Application to the Circuit/Hypergraph Partitioning Problem. In *Proc. 4th of International Symposium on Physical Design*, April, 2000.

[23] J. Paredaens. Spatial databases, the final frontier. In G. Gottlob and M.Y. Vardi, editors, *In Proceedings of International Conference on Database Theory*, volume 893 of *Lecture Notes in Computer Science*, pages 14–32. Springer-Verlag, 1995.

[24] R. Ramakrishnan. *Database Management Systems.* McGraw-Hill, 1998.

[25] U. Ramer. An Iterative Procedure for the Polygonal Approximation of Planar Curves. *Computer Graphics and Image Processing*, 1:244–256, 1972.

[26] P. Revesz, R. Chen, and M. Ouyang. A Software Architecture for Constraint Database Systems. In *Software Architectures, Components, and Frameworks*, submitted.

[27] P. Z. Revesz. Constraint Databases: A Survey. In L. Libkin and B. Thalheim, editors, *Semantics in Databases*, number 1358 in LNCS. Springer-Verlag, 1998.

[28] P. Z. Revesz and Y. Li. MLPQ: A Linear Constraint Database System with Aggregate Operations. In *Proc. 1st International Database Engineering and Applications Symposium*, 1997.

[29] A. Silberschatz, H. Korth, and S. Sudarshan. *Database System Concepts.* McGraw-Hill Press, 1998.

[30] W. R. Tobler. A Continuous Transformation Useful for Districting. *Annals, New York Academy of Sciences*, 219:215–220, 1973.

[31] W. R. Tobler. Pseudo-Cartograms. *The American Cartographer*, 13(1):43–50, 1986.

[32] J. Ullman. *Principles of Database and Knowledge-Base Systems*, volume I and II. Computer Science Press, 1988-1989.

[33] L. Vandeurzen, M. Gyssens, and D. Van Gucht. On the desirability and limitations of linear spatial query languages. In M. J. Egenhofer and J. R.

Herring, editors, *Proceedings of the 4th International Symposium on Spatial Databases*, volume 951 of *Lecture Notes in Computer Science*, pages 14–28, Berlin, 1995. Springer-Verlag.

[34] B. White, I. Gregory, and H. Southall H. Analyzing and Visualising long-term Change . In *www.geog.qmw.ac.uk/gbhgis/gisruk98.*

[35] E. Wong and K. Youssefi. Decomposion - A Strategy for Query Processing. *ACM Transactions on Database Systems*, 1(3):223–241, 1976.

[36] M. F. Worboys. *GIS: A Computing Perspective.* Taylor&Francis, 1995.