A Multidimensional Spreadsheet-Style Browser

by

Feng Wang

B.Sc., Shandong University, 1986

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

We accept this thesis as conforming
to the required standard

[redacted]

Dr. William W.Wadge, Supervisor (Department of Computer Science)

[redacted]

Dr. Michael Fellows, Departmental Member (Department of Computer Science)

[redacted]

Dr. Eric Manning, Outside Member (Dept. of Elec. & Comp. Engineering)

[redacted]

Dr. Bjorn Freeman-Benson, External Examiner (Carleton Univ., Ottawa)

Supervisor: Professor William W.Wadge

ABSTRACT

In this thesis, we present a multidimensional spreadsheet-style browser. This browser allows users to visualize and analyze multidimensional data sets, in particular multidimensional output data produced by Granular LUCID (GLU). As GLU has simplified the complex task of generating multidimensional data, the need for an easy and efficient means of viewing these data becomes more evident.

GLU is a hybrid of the LUCID and C languages. LUCID is the declarative nucleus of GLU. Because LUCID is a functional, dataflow language for processing multidimensional data, the traditional imperative approach to I/O is not appropriate. Instead, a multidimensional browser can be used to display and analyze these multidimensional output data sets generated by GLU.

The approach we develop for browsing the output of multidimensional GLU programs is a three step process. First, all output data of the multidimensional GLU program are collected in a multidimensional warehouse file. Second, all data in this warehouse are read into memory. Finally, users browse these data from a variety of viewpoints.

In addition to a tabular view of the multidimensional data set, the browser provides several analysis functions. Users select data for further investigation by mapping two chosen dimensions to the horizontal and vertical axis of the display and fixing all other dimensions. Given these parameters, the browser can generate a spreadsheet view according to the user's need. For dimensions with a large coordinate space, users can zoom in a particular portion of the display for combination and comparison. Furthermore, common mathematical functions such as SUM, AVE, MIN, MAX can be performed. These types of features are often supported by on-line analytical processing (OLAP) products.

Like any OLAP product, this browser provides three basic functions - slicing and dicing, comparison and combination of rows and columns, and analytical calculations. In

order to effectively achieve these functions for browsing multidimensional data our design involves careful consideration of a storage structure, storage order and data access address scheme. Attention to these key elements yields a user friendly browser for use with any multidimensional data warehouse of similar structure.

Examiners

[redacted]

Dr. William W.Wadge, Supervisor (Department of Computer Science)

[redacted]

Dr. Michael Fellows, Departmental Member (Department of Computer Science)

[redacted]

Dr. Eric Manning, Outside Member (Dept. of Elec. & Comp. Engineering)

[redacted]

Dr. Bjorn Freeman-Benson, External Examiner (Carleton Univ., Ottawa)

Table of Contents

List of Tables

List of Figures

Acknowledgments

I would like to take this opportunity to thank my supervisor, Dr. William W. Wadge, for his advice during my whole research and study of M. Sc. program at the University of Victoria. Dr. William W.Wadge suggested this research topic. I am grateful to him for his continuous guidance, encouragement and seemingly endless patience throughout the development of this thesis. I am also grateful to him for his financial support which made it possible to finish my M. Sc. program and this thesis.

In addition, I would like to thank Dr. Eric Manning, Dr. Michael Fellows, and Dr. Bjorn Freeman-Benson for serving on my supervisory committee, kindly assisting and patiently reviewing this research throughout my M. Sc. program.

Furthermore, I would also like to thank Mr. Thomas W. Barr for correcting my English and providing valuable suggestions.

Finally, I would like to thank my wife, my sister and my parents for their help, encouragement and support.

Chapter 1

Introduction

In an age in which information is becoming a commodity, the ability to gain access to and quantify vast data sets is a key element for making effective use of an enterprise's data. A common approach to data acquisition, analysis and storage is through the use of an SQL (structured query language) interface to a relational database engine. SQL has its limitations; several motivating factors, such as the need for data visualization flexibility, better performance and ease of analysis have stirred up interest in alternate methodologies. In an effort to minimize disk storage, improve performance and increase flexibility, internal architectural database designs which facilitate multidimensional data are being deployed. In this thesis, to assist the user in viewing and analyzing the multidimensional data generated by Granular LUCID (GLU) programs, we will concentrate on the design of a browser which simplifies data quantification as well as analysis based on on-line analytical processing (OLAP). The power of a multidimensional warehouse can be tapped through the effective use of a browser capable of organizing views and performing calculations. In addition to browsing multidimensional data, the implementation of a multidimensional warehouse will also be explored. We will provide background on these aspects and components of multidimensional data warehouses in later chapters.

As data is accumulated, the need arises to give meaning to, and summarize the overall trends and/or conclusions. As one reviews complex data, several requirements or criteria are helpful to assist the end-user in correctly analyzing these data. These include

the ability to

- create multidimensional views of data
- access many different types of files
- experiment with various data formats and aggregations
- define and animate new information models
- summarize, consolidate, sum and apply formulae to these models
- drill down, roll up, slice and dice, pivot, and rotate consolidation paths

The significance of the above items (from [CCS93a]) will become apparent as later chapters are read.

1.1 Overview of GLU

One of the means of generating multidimensional data is by using GLU. GLU, a multidimensional declarative programming language, is a combination of LUCID and the C language. The basic idea of GLU is to use LUCID as a language for composing applications implemented as C functions. A GLU program is a LUCID program in which user-defined functions can be C functions and values can be (almost) any valid C data structure. In other words, a program in GLU consists of two distinct parts: a LUCID part that specifies the program composition and a C part that defines various data types and data functions referred to in the LUCID part.

LUCID, the heart of GLU, is a pure dataflow language in which a program is expressed as a structured set of multidimensional data dependencies. LUCID was developed in 1976 by Ashcroft and Wadge [WA85] as a system for writing and proving properties about programs. Not only is LUCID non-procedural, it is dynamic with respect to its view of computation. This unusual aspect is characteristic of dataflow. In other words, data is generated or consumed by stationary operations, while it is in motion. In addition, the declarative nature of LUCID allows exact and easy composition of problem solu-

tions. LUCID programs are incompatible with the conventional imperative model of programming. In particular, LUCID programs do not execute efficiently on control-driven processors. Over the past ten years, the LUCID language has evolved from a temporal dataflow language to a multidimensional dataflow language. Here, we briefly review the essential aspects of the LUCID part of GLU.

A LUCID program is a structured set of equations where each equation defines a variable. The left-hand-side (LHS) of an equation is a variable. The right-hand-side (RHS) of each equation can be a constant, a variable, an operation on other terms, or a user defined function reference. Associated with each set of equations is one or more user-defined dimensions. These dimensions are orthogonal and infinite in extent. A variable denotes a scalar value at each point in the space. As a result, a LUCID program computes values of specific variables at specific points in the nested multidimensional space that it defines. Computation of each desired value requires the computation of other values at various points as specified by the equations of the LUCID program. In an Original LUCID program [WA85], however, a variable or an expression denotes an infinite sequence of values that is thought of as a temporal sequence in 1 dimension: time.

In GLU, we do not have to think of a variable as being an infinite object (an infinite sequence laid out in many dimensions). Rather, we can think of it as a simple object that depends on the positions in the multidimensional space. In other words, different positions in different dimensions can generate different values for a specific variable. In addition, the demand-driven dataflow computation model, namely *eduction,* is used to evaluate GLU programs. We will further explain this model in chapter 2.

1.2 Motivation

Current implementations of GLU only navigate in the time dimension. In fact, at the top most level time is the only dimension. GLU programmers can declare other dimensions, but only locally - at the outer level the inner dimensions are all set to 0. Thus, a problem with GLU is extracting the input dimensions and relations among these dimensions when generating a series of multidimensional output data. The following example, test.g, illustrates this problem.

```
test.g

((sum@.d1 d1_pos)@.d2 d2_pos)@.d3 d3_pos
where
  dimension d1, d2, d3;
  d1_pos = #.time % 4;
  d2_pos = (#.time / 4 ) % 5;
  d3_unbound = #.time / 20;
  d3_pos = if (d3_unbound < 7)
           then d3_unbound else eod fi;
   sum = #.d1 + #.d2*10 + #.d3*100;
end
```

Figure 1.1 An example of a GLU program

This example has three dimensions (d1, d2, d3), a variable **sum** defined by a summation formula at different dimensional positions with a boundary condition. In this example, we do not think of **sum** as being an infinite object (an infinite sequence laid out in dimensions) but as being a simple object dependent on the position in three dimensions. In other words, different input dimensional positions can generate different result data. After compiling and executing this program, the following output data are generated.

```
[time: [values]
[0]:  0
[1]:  1
[2]:  2
[3]:  3
```

```
.....................

   [10]:  22
   [11]:  23
   [12]:  30
   [13]:  31
   [14]:  32
.....................

   [136]:  640
   [137]:  641
   [138]:  642
   [139]:  643
```

Difficulties arise when determining which input dimensional positions (which values of d1, d2 and d3) were used in generating these result values. The default driver for GLU programs only navigates through one dimension: time. In fact, different dimensional positions will generate different resulting values. In order to get these values and the corresponding coordinates, therefore, the programmer needs to navigate the three dimensional space with some sort of browser program. For the above example, we can modify program test.g and write a browser program test.c as follows:

test.g

```
local int browser(int, int, int, int);

((sum@.d1 d1_pos)@.d2 d2_pos)@.d3 d3_pos
where
    dimension d1, d2, d3;
    d1_pos = #.time % 4;
    d2_pos = (#.time / 4 ) % 5;
    d3_unbound = #.time / 20;
```

```
    d3_pos = if (d3_unbound < 7) then d3_unbound else eod fi;
    sum = browser(g1, g2, g3, value);
    g1 = #.d1;
    g2 = #.d2;
    g3 = #.d3;
     value = #.d1 + #.d2*10 + #.d3*100;
end
```

test.c

```
#include "test_glu.h"

    int  browser(int d1, int d2, int d3, int ss)
    {
    printf("[%d, %d, %d] = %d", d1, d2, d3, ss);
    return(ss);
}
```

Figure 1.2 An example program to navigate dimensional positions in GLU

After compiling using "gc test.g test.c" and running "./test", the following output data are generated:

```
[dim. pos.] = values
[0, 0, 0] = 0
[1, 0, 0] = 1
[2, 0, 0] = 2
[3, 0, 0] = 3
.....................
.....................
.....................
[3, 3, 6] = 633
[0, 4, 6] = 640
[1, 4, 6] = 641
[2, 4, 6] = 642
[3, 4, 6] = 643
```

Although we can now see which coordinates are associated with each output data item, it is still not a user friendly interface and is inadequate for visualizing and analyzing large multidimensional data sets. In particular, the user has no control over the order in which data are displayed, no choice as to which data to view, and no ability to aggregate and summarize the data. Therefore, it is necessary to have a tool to intuitively navigate, display and analyze these multidimensional output data. This tool could be a graphics package, multidimensional database, query language or some other kind of display software. In this thesis, a multidimensional spreadsheet-style browser is adopted because it is simple to implement and provides an intuitive approach to displaying and analyzing multidimensional output data. In order to use this browser, we first collect all these multidimensional output data into a warehouse, then this multidimensional spreadsheet-style browser can provide a user friendly interface for viewing and analyzing these multidimensional output data of GLU programs. This is the motivation and the basic idea of the thesis.

The problems GLU users have with output are just an example of the general problem of output for functional languages.As Jones and Wadler says in [JW93]:

> Input/output has always appeared to be one of the less satisfactory features of purely functional languages: fitting action into the paradigm feels like fitting a square block into a round hole.

The problem is that functional languages are lazy (demand-driven), but most computer output is eager (data-driven). The obvious answer is to make output demand-driven as well, but it is not practical to make the user ask for individual data items one by one. A browser solves this problem by giving the user a higher level interface in which to express their demands. The browser software translates the users actions (e.g. mouse clicks) into demands for individual data items.

In a sense, we are adapting some new ideas in industrial software to solve a problem that academics have been working on for many years.

Originally, we wanted to attach the browser directly to the running GLU programs but serious technical problems (to be discussed later) prevented this. Nevertheless, the browser can still be used to view GLU output, and if the GLU system is ever redesigned with a more open software architecture, the browser could be easily adapted to give the first complete solution to the problem of functional language output.

1.3 OLAP (on-line analytical processing)

OLAP is a relatively new concept in the database world. OLAP is the name given to the dynamic analysis required to create, manipulate, animate and synthesize information from different data analysis models [CC94]. OLAP lays the groundwork for flexible analysis and is gaining momentum within the database community.

In fact, there has been an explosion of interest in multi-dimensional data visualization since I began graduate studies, and especially in the last year. There are now a number of extremely sophisticated products on the market. Recently, for example, (in December 1995) the SAS institute announced their new SAS/SPECTRAVIEW tool which enhances data warehousing, for example, by permitting four dimensional exploitation of the SAS data warehouse. In the announcement in the SAS web site (http://www.sas.com/corp/corporate/preleases/122095/news1.html), they explain:

"For the business professional, using SAS/SPECTRAVIEW software is like flying in a helicopter and wearing x-ray spectacles at the same time,". "You can visualize enormous volumes of multi-dimensional data in their entirety to see overall patterns; and then slice and dice that mass of data from any direction to reveal key internal trends and relationships."

An integrated component of the SAS System -- a suite of information delivery software for business decision making -- SAS/SPECTRAVIEW software allows business professionals to visualize up to four dimensions of data at a time. In addition to displaying data in a three-dimensional space, the user can define colors for different values of a chosen characteristic (the 'response variable') to simulate a fourth dimension. The effects of a fifth variable -- time -- can be seen by running animations of the data. "SAS/SPECTRAVIEW software provides decision makers with a very fast and intuitive way to identify the pri-

mary factors that cause change."

In this thesis, we do not try to implement a full featured OLAP tool. Our purpose in this thesis is to provide a simple, portable, general purpose tool for browsing multidimensional GLU output.

It is our intention to incorporate some basic functionality of OLAP within our browser. One of the important characteristics of OLAP is multidimensional analysis. One way to achieve conformity to OLAP would be to adhere to the 12 rules as laid down by E.F. Codd in "Beyond Decision Support" [CCS93a]. Total conformity may not be achieved. However, providing base functionality and support for some of the rules would be advantageous. Chapter 5 explores this in detail.

1.4 Multidimensional spreadsheet-style browsers

1.4.1 Spreadsheets

Spreadsheet software has become very popular and is widely used on personal computers as well as on workstations. The first electronic spreadsheet was developed in 1978. Spreadsheet programs can be used as electronic replacements for traditional financial modeling tools. Generally, a spreadsheet is organized as a grid of rows and columns in a window. Each intersection of a row and column is an entry position **(cell).** Each cell can hold information. All the information is visible and the user's operations are interactive. In a spreadsheet, the computation is always up-to-date because the display is re-evaluated continuously when the user modifies the contents of a cell. Today's spreadsheets can support powerful calculation capabilities, powerful functions, integrated business graphics, word processing and database management.

When using a spreadsheet, the computer's screen becomes a two-dimensional dis-

play window. The user can scroll this window in all four directions to look at any part of the sheet. The total number of rows and columns that a spreadsheet may have is dependent on the design of the spreadsheet and the computer resources available. The display width of a column can be one character wide or wider, based on the user's needs. The screen generally consists of two basic areas: the display window area and the control area. The display window shows the displayed cell grid and data while the control area is used for data input and interactive system messages.

1.4.2 A multidimensional spreadsheet-style browser

Expanding the basic spreadsheet model to accommodate multidimensional data while preserving ease of use yields a multidimensional spreadsheet-style browser. A standard spreadsheet style browser provides a read-only view of two dimensional data sets. With the introduction of multidimensional data warehouses into today's business world a more sophisticated browser is necessary. In order to view multidimensional data easily and efficiently a multidimensional spreadsheet-style browser becomes an ideal tool to intuitively display and analyze multidimensional warehouse data. Based on this consideration, our browser combines display and analysis ability as well as allowing access to complex multidimensional data. We have seen that output data of GLU is in general multidimensional. By using this multidimensional spreadsheet-style browser users can view selected GLU output from different dimensional viewpoints, as well as perform calculations on these multidimensional data. In addition, this multidimensional spreadsheet-style browser is not limited to data produced by GLU. Other warehouses may also be displayed and analyzed as long as the format is known by our browser.

The organization of this thesis is as follows: Chapter 2 gives the necessary background for multidimensional programming using GLU, multidimensional spreadsheet-style browser, and OLAP. Chapter 3 describes the design and implementation of a multi-

dimensional spreadsheet-style browser. Chapter 4 describes the verification and use of the multidimensional spreadsheet-style browser. Chapter 5 explains in detail our browser's relation to OLAP. Chapter 6 formulates a conclusion and discusses future work.

Background

2.1 Some background on GLU

2.1.1 What GLU is

GLU is described in [R. Jagannathan, C. Dodd, *GLU Programmer's Guide(version 0.9)*, 1994, p1, p3]:

> GLU is a very-high-level system for constructing and executing high-performance applications for diverse platforms ranging from heterogeneous workstations to shared-memory multiprocessors to massively parallel processors.
>
> In GLU, a high-performance application is a LUCID composition of sequential functions expressed in the C language. Existing sequential applications can be converted into the GLU framework easily by reusing the C code fragments with the addition of a small amount of code.
>
> GLU could be termed a Procedural Extension to LUCID. GLU is a simple yet powerful extension of LUCID: user-defined functions in GLU can be defined procedurally and values they consume and produce can be complex data structures.

In general, programming in GLU is declarative (rule-based) programming, and it involves new ways of thinking about programming because traditional procedural programming is imperative, not declarative. There are two ways to think of the GLU model of programming. The first way is a combination of declarative (LUCID) programming with imperative (C) programming. Programmers can usually specify *what* to compute in the declarative (LUCID) programming and *how* to compute it in the imperative (C) programming. The second way to think of GLU is as a procedurally-extended declarative programming language. In other words, GLU is a declarative programming language, but with imperative C extensions. Therefore, we can say that the first way of thinking is good for those who are familiar with procedural programming and the latter is good for those who are familiar with declarative programming.

2.1.2 Syntax of GLU

The LUCID language has evolved from a temporal dataflow language to a multidimensional dataflow language. It is the only dataflow language with multidimensional data structures. Therefore, we pay special attention to its multidimensional features when discussing the syntax of GLU. The discussion here is based on [R. Jagannathan, *Coarse-Grain Dataflow Programming of Conventional Parallel Computers,* IEEE Computer Society, April 1995, p3] with some modifications.

A LUCID program is a structured set of equations where each equation defines a variable. The left-hand-side (LHS) of an equation is a variable. The right-hand-side (RHS) of each equation is a term which can be a constant, a variable, an operation applied to other terms, or a user-defined function whose arguments are other terms.

1. If the RHS is a constant, the value of the LHS variable at all points in the enclosing multidimensional space is that constant. For example, the equation **pi = 3.14159267** defines **pi** at all points to be the constant 3.14159267 in the enclosing multidimensional space.
2. If the RHS is a variable, the value of the LHS variable is the same as that of variable on the RHS at all points in the enclosing multidimensional space. For example, **a = b; b =3**. The value of the **a** variable is the same as variable **b,** which is the constant 3 at all points in the enclosing multidimensional space.
3. If the RHS is an operation applied to other terms, the value at each point of the LHS variable is given by the application of the operation to values of the operand terms at appropriate points in the enclosing multidimensional space.

   a). If the operation is a data operation such as +, or <, the value of the LHS variable at a given point is the result of the operation applied to the appropriate values of the operand terms at the same point. For example, the equation **a**

= if p then b else c fi defines **a** at each point to be either the value of **b** or **c** at that point depending on whether the value of **p** is **true** or **false** at that point.

b). If the operation is a dimensional operation, then the value of the LHS variable at a given point is the result of the operation applied to values of the operand terms at points defined by the operation.

The two basic dimensional operations are @ and #. The expression **a @.x n** refers to the value of **a** at the $\mathbf{n}^{th}$ point in dimension **x**. The expression **#.x** refers to the "current" implicit point of evaluation in the **x** dimension. Other dimensional operations can be defined in terms of @ and #. For example, the dimensional operation **fby.x** which should be read as ("followed by in the **x** dimension"), as in **a fby.x b**, can be defined as follows:

```
if( #.x == 0 ) then a @.x 0 else b @.x (#.x - 1) fi;
```

In the equation, **a = b fby.x c**, for all points where the x dimension position is 0, the value of **a** is the value of **c** at the preceding point (in the **x** dimension).

Also, dimensional operation **asa.x** (which should be read as "as soon as in the **x** dimension"), in expression **e asa.x p**, can be defined as follows:

```
e asa.x p = a
where
   a = if (p @.x #.x ) then e @.x #.x else next.x a fi;
      where
      next.x a = a @.x (#.x + 1);
      end;
end
```

In the equation, **a = e asa.x p**, the value of **a** at all points in the **x** dimension is the value of **e** at a point in the **x** dimension such that the value of **p** at that point is **true** and the value of **p** at each of the preceding points in the **x** dimension is **false**.

4. If the RHS is a user-defined function, the value of the LHS variable is the same as the value of invoking the RHS function on its argument expressions using call-by-need semantics.

```
a = nxt( b, n )
where
   nxt( c, n ) = c @.x (#.x + n );
end
```

The value of **a** at a given point in the **x** dimension is the value of function **nxt** given argument **b** and **n** at the same point. Function **nxt** is the value of **b, n** points from the current point in the **x** dimension.

In LUCID, user-defined functions like operations can be dimensionally abstract. For example, it is possible to define the function **nxt** such that it works in any dimension rather than just in the **x** dimension.

```
a = nxt.x( b, n )
where
   nxt.z( c, n ) = c @.z (#.z + n );
end
```

After discussing these basic syntax rules, we explore how to evaluate the programs in GLU.

2.1.3 *Eduction* and I/O in GLU

In GLU, LUCID programs are evaluated using a special dataflow computation model, namely, *eduction.* Eduction is a demand-driven dataflow computing model. The word *eduction* is explained in the Oxford English Dictionary as follows. "*The action of drawing forth, eliciting, or developing from a state of latent, rudimentary, or potential existence; the action of educing (principles, results of calculations) from the data.*"

The eduction model can be described as follows in [Ashcroft, Faustini, Jagannathan, Wadge, *Multidimensional Programming,* 1995, p90:

1. Need for a result by the outside world causes it to be demanded.

2. If a value e at context [c] is demanded at some stage, then and only then are values demanded that are known to directly determine value e at context [c].

Now we give an example demonstrating how to evaluate GLU programs using the eduction model.

```
result = ( if flag then a else b fi ) * x
```

Demand for the value of $\mathbf{result}_{[i]}$ (value of **result** at context [i]) causes demands for values of $\mathbf{flag}_{[i]}$ and $\mathbf{x}_{[i]}$ because only these values are needed to determine the value $\mathbf{result}_{[i]}$. If the value $\mathbf{flag}_{[i]}$ is defined and is **TRUE**, the value $\mathbf{a}_{[i]}$ can be demanded subsequently because this value is now needed to determine value of $\mathbf{result}_{[i]}$. Then, the value of $\mathbf{result}_{[i]}$ is the value of $\mathbf{a}_{[i]}$. On the other hand, If the value $\mathbf{flag}_{[i]}$ is defined and is **False**, the value $\mathbf{b}_{[i]}$ can be demanded subsequently since this value is now needed to determine the value of $\mathbf{result}_{[i]}$. Then, the value of $\mathbf{result}_{[i]}$ is the value of $\mathbf{b}_{[i]}$.

In fact, the eduction model does only useful computation because it does not compute values when these values are not needed for a result. In the above example, observe that eduction avoids the computation of value $\mathbf{b}_{[i]}$, which is not necessary when $\mathbf{flag}_{[i]}$ is **TRUE**. On the other hand, eduction avoids the computation of value $\mathbf{a}_{[i]}$, which is not necessary when $\mathbf{flag}_{[i]}$ is **FALSE**.

The eduction computation model is demand-driven. Its main advantage is to save resources because it only evaluates values as needed. However, result values depend on specific contexts for different dimensions, i.e. different contexts can generate different result values. In the above example, the value of **result** depends on context [i]. Because LUCID is a dataflow language with multidimensional data structures, the value of **result** may depend on several contexts for different dimensions. In the following, we further explain why we need a browser for GLU output.

First, it is necessary to look at how GLU handles I/O. The demand for output values propagates through a program producing a demand for input. Output from a GLU program can be thought of as a stream. The interpreter applies a dynamic interpretation of LUCID sequences to a program's input and output streams. The program and interpreter can be thought of as a filter with the program supplying the rules to the interpreter. As GLU evaluates a program, it begins by computing the output at time t=0. The newly computed value is placed in the output stream. As processing continues, values are computed and placed in the output stream for time t=1, 2,..., n. As an example, consider the following:

```
s@.d #.time
where
  dimension d;
   s = 0 fby.d s + 10;
end
```

The initial computation in the above example will produce 0 as output. Further computations will yield 10, 20, 30 and so forth. It should also be noted that this output stream has one dimension. This single dimension is time. In addition, because a variable is defined based on a dimensional **where** clause and this variable and this dimensional **where** clause can vary in many dimensions, the problem is how to make the inner dimensionality compatible with the outer dimensionality. That is, which of the values associated with the extra dimensionality is used to give meaning to the outside object? In order to solve this problem, Indexical LUCID sets all new dimensions made by the dimension declaration of the **where** clause to 0. Therefore, all dimensions are set to 0 except time for output in Indexical LUCID.

After reviewing the above discussion, one may wonder why a multidimensional

GLU program just produces one dimensional output. The reason relates to limitations of viewing the output data. For example, the physical size of a screen is a limitation which creates difficulty in being able to see enough data so as to correctly comprehend the actual trend or variation of the output values. If the data comes from a multidimensional data set, the user must traverse this data. The path selected is as long as the session that is producing output and may easily exceed screen size limitations. In addition there is no best path that is going to work for all output and all users. It is up to the user to navigate through the data space. GLU's built-in output only provides one dimensional output streams. The user has the difficult task of creating multidimensional output by developing custom software using C.

When there is only one dimension to consider, it is a simple task to map output into a "real-world" time dimension. When there is more than one data dimension it becomes difficult to map a one dimensional output stream into the true multidimensional structure that GLU has easily manipulated. The problem is further complicated when these extra dimensions are unbounded. Even in the case of an internal time and space dimension, there is not a "best" or "optimal" way to produce output data without any guidance from the user. It is possible to output multiple streams if the space dimension is limited. This is often not the case for much of the relevant corporate world data requirements.

The development of a data-driven model for multi-dimensional output has yet to be seen. In a data-driven model, every time an output command is executed, the program "pushes" data into the output stream. The user simply can not keep up. With unbounded multidimensional output data, this just does not work. These problems provide motivation for a browser. A browser which accepts interactive instructions from the user, allows one to describe which parts of the data are to be viewed. Without a browser, the program

would have to prompt the user for detailed information about output format and structure, making for very complicated program logic. For solving this problem, we need a software tool to view the output data at different contexts for different dimensions. Table 2.1 compares the internal structure and I/O of traditional UNIX programs, Original LUCID and GLU. In the next paragraph, we will further discuss multidimensional programming in GLU.

	Unix	Original LUCID	GLU
Number of dimensions of data	One dimensional	One dimensional	Multi-dimensional
I/O	Data-driven	Demand-driven	Demand-driven
Mapping between real world and output	Yes	Yes	No. a browser is needed to make demand-driven output practical

Table 2.1. Comparison of the internal structure and I/O

2.1.4 Multidimensional programming in GLU

LUCID was developed in 1976 by Wadge and Ashcroft [WA85] as a system for writing and proving properties of programs. Original LUCID (the language of [WA85]) just supports one dimension; time. The most unusual aspect of LUCID, in addition to being non-procedural (declarative), is its dynamic view of computation. In other words, data is in motion and is generated and consumed by stationary operations and functions. Because of this view, commonly known as dataflow, Original LUCID can be considered to be one of the first dataflow programming languages.

In an Original LUCID program, a variable or an expression denotes an infinite sequence of values that is thought of as a temporal sequence. The basic Original LUCID operators **first, next,** and **fby** (for "followed by") are used to operate on this sequence. It is also possible to specify subcomputations in the language by using nested time. In particular, a variable can be defined to vary in a nested time dimension so that variables

defined in outer time dimensions appear to be constant.

Since we want to deal with multidimensional data it should be noted that Original LUCID does not naturally support multidimensional data structures such as arrays and trees. These data structures are quite common in business and scientific applications. This omission was the basis for forming GLU from Original LUCID by incorporating multidimensional capabilities. In the following paragraphs we present a generalization of Original LUCID for multidimensional problem solving and show its expressiveness towards the multidimensional domain. After a review of Original LUCID, GLU's user-defined dimensions and dimensional abstraction will be introduced.

It is important to have a clear understanding of Original LUCID in order to fully appreciate GLU. Let us explore Original LUCID by understanding the key points of a procedural language and compare that with Original LUCID's non-procedural approach. In a procedural language, a variable is associated with a sequence of values. The initial assignment of a variable is the first time program flow encounters this variable. When control moves through the procedural program, each assignment that is encountered modifies the memory location associated with the variable [FJ93]. If we were to trace the values associated with a particular variable, we could generate a historic sequence of values which had been assigned to that variable. In a conventional procedural language there is usually no trace or history of the value associated with a variable because assignment is destructive. Original LUCID variables, however, are not associated with destructively modifiable memory locations. Like its procedural counterpart, an Original LUCID variable takes on different values at different times during the computation. The "history" or sequence of values associated with an Original LUCID variable is achieved through equational definitions rather than destructive assignment and control flow. The equation **x = 1** defines the variable **x** to be a constant. (i.e. **x** is equal to 1 at all times with program execution). On the other hand **y = 1 fby 2 fby 3** is an equation that says that the

initial value of **y** is 1 at **time = 0**, the next (**time = 1**) value of **y** is 2 and at all times, beyond this, the value of **y** is always 3. The key point of Original LUCID is that there is no destructive assignment so that the variable history is always available. Not only is the current value of a variable available for assignment purposes on the RHS of an equation, but previous values of that variable can also be used as well. This is the main motivation for maintaining the variable history.

The following expression uses the idea of an Original LUCID variable taking on different values at different times. The pair of equations given below define the variable **odd:**

```
first odd = 1
next odd = odd + 2
```

The first equation defines the first or initial value of the odd stream and second equation the successive value of odd in terms of the current value of odd plus two.

We can think of the above equations as equivalent to the following recurrence relation:

$t = 0 \quad odd_t = 1$

$t > 0 \quad odd_{t+1} = odd_t + 2$

The **t** in the recurrence relation is the implicit dimension or time context of Original LUCID. Note that there is no explicit mention of **t** or **time** other than the first equation which states that the initial value of **odd** is 1. We are still able to generate the stream of values <1, 3, 5, 7,...> without the explicit use of indices. (In this sense, Original LUCID captures time implicitly during the iteration.) Consequently, a multidimensional problem would have to be solved using time and some form of monolithic data structure such as a list or string. The introduction of multidimensional indexing into LUCID yields indexical LUCID which allows for the expression of multidimensional problems without the use of monolithic data structures. The latter has important meanings for parallel and

distributed implementations. Multidimensional problems can be found in many application areas and specially in scientific computing and computer graphics.

2.1.5 Indexical LUCID

The following discussions give some necessary details of indexical LUCID; three examples are used to illustrate these points. These examples originally appeared in [Faustini, Jagannathan, *"Multidimensional Problem Solving in Lucid,"* 1993, p4, p5, p8] but have been modified.

Several new constructs are provided by Indexical LUCID. The user explicitly declares new dimensions using a **dimension** statement as needed. In addition, a dimensional **where** clause is introduced for the declaration of new or equivalent dimension names.

An example is given as follows and it contains a dimensional **where** clause with two new dimension names **vertical** and **horizontal**.

```
x
where
        dimension vertical, horizontal;
        x = ...
end
```

In the above program the variable **x** may be defined to change over the two new dimensions **vertical** and **horizontal**. To determine how **x** varies with either of the dimensions **vertical** and **horizontal** we need to examine the actual definitions of **x**. For example, if **x = 3**; then **x** is a constant 3, if **x = 1 fby x+1**; then **x** varies with dimension **time** like <1, 2, 3...>. To make **x** vary with dimensions **vertical** or **horizontal**, we introduce operators specific to the new dimension names. All of the usual Original LUCID operators can be extended to operate on any dimension name in a simple and natural manner. For example, the infix binary operator **fby** is extended to work on the **vertical** dimension

through use of the operator **fby.vertical**. In a similar manner all of the usual Original LUCID operators can be extended to work on any dimension name if this dimension name has been previously introduced by a dimension declaration.

In addition, dimensional locators are an important concept of Indexical LUCID. Each dimension name introduced by a dimension declaration also introduces a new dimensional locator having the same name as the dimension name. For example, the dimensional locator **vertical** can be used in the expression part of a program and the value it takes depends on when it is used. In general, the value of **vertical** at position *i* in the **vertical** "dimension" is *i*. In other words, dimensional locators tell the programmer where they are in the dimensional space that has been built up with the dimensional **where** clauses. Note that all dimensions in Indexical LUCID vary over the integer set (hence the name Indexical LUCID).

Now we discuss dimensional abstraction. In Original LUCID the **time** dimension is absolute, therefore, ordinary function definition is adequate. So we do not need abstract dimension names in function definitions. In Indexical LUCID, however, we need abstract dimension names in function definitions. For example, the function **VectorSum** sums the first 200 elements along the x dimension:

```
.......
where
     dimension x, y;
     VectorSum(V1) = sum asa.x x>200
        where
           sum = 0 fby.x V1+sum;
        end;
     A = 1.1 fby.x A + 5;
     B = 0.8 fby.y B + 7;
     sum1 = VectorSum(A);
     sum2 = VectorSum(B);
........
```

```
end
```

The problem with the above definition of **VectorSum** is that it will only compute the sum of the first 200 elements along the **x** dimension. Thus the value of **sum1** will be as expected and the value of **sum2** will be **200*B** because **B** does not vary with respect to the **x** dimension. In order to abstract dimensional names so that we can write one vector sum function for any dimension, we need to augment the way in which Original LUCID defines and uses functions. Indexical LUCID augments Original LUCID function definitions by permitting the declaration of formal dimensional names on the left hand side of the equation. To be more precise a function name may be followed by a sequence of formal dimension names each prefixed with a period (.). The following is a rewritten version of the vector sum example with both dimensional parameters and value parameters:

```
.......
where
     dimension x, y;
     ......
     VectorSum.k(V1) = sum asa.k k>200
        where
           sum = 0 fby.k V1 + sum;
        end;
     A = 1.1 fby.x A + 5;
     B = 0.8 fby.y B + 7;
     sum1 = VectorSum.x(A);
     sum2 = VectorSum.y(B);
     .......
end
```

The k dimension is a formal dimension name. The actual dimension name is determined when the function **VectorSum** is used, as in **VectorSum.x(A)** or **VectorSum.y(B)**. In the first of these, the actual dimension name is **x**; this name is used in the actual computation of the vector sum by the function **VectorSum**.

With the above description of Indexical LUCID, we can now give an example to illustrate its expressiveness in solving multidimensional problems.

This example shows array computations which refer to nested computations that are used to manipulate multidimensional arrays. Because an array is a multidimensional data structure, we give an example to explain it. This example was originally written in [Ashcroft, Faustini, Jagannathan, Wadge, *Multidimensional Programming,* 1995, p32] and is presented here with some modifications.

For computing the distribution of an electric potential on a two-dimensional grid over time with fixed potential at the electrodes, we use a successive over-relaxation method. This can be implemented by the following program.

```
grid
 where
 dimension v, h;
 grid = if ELECTRODE then POTENTIAL else 0 fby avg( grid )
  where
  ave( g ) = (prev.v g + prev.h g + next.v g + next.h g ) /4;
  end;
 end
```

There are two dimensions (**v**: vertical and **h**: horizontal) in the **where** clause. The value of the grid at this point is always **POTENTIAL** if the **v** and **h** positions denote an electrode. Otherwise, it starts off as 0 and at each successive **time** dimension position, it is the average of the four neighboring values of grid at the previous **time** dimension position that is used. The operators **prev** and **next** are defined as: `prev.v x = x @.v (v-1)` and `next.v x = x @.v (v+1)`. Furthermore, we can also make it work equally well in 3-dimensions by simply adding one dimension (**d**: depth) which makes the program look like this:

```
grid
 where
 dimension v, h, d;
```

```
grid = if ELECTRODE then POTENTIAL else 0 fby avg( grid )
  where
  ave( g ) = (prev.v g + prev.h g + prev.d +
              next.v g + next.h g + next.d g) /6;
  end;
end
```

Finally, GLU not only has declarative and parallel features but it also supports C language objects. GLU is a functional dataflow language for processing multidimensional data and solving multidimensional problems. However, GLU's built-in output has only one dimension (time), therefore, it is not possible, using built-in output, to see how data varies in more than one dimension. In order to solve this problem, we designed and implemented a multidimensional spreadsheet-style browser. Next, we present some basic background about spreadsheet programs and how they apply to our browser.

2.2 Spreadsheets

One of the first applications to gain widespread popularity during the explosive introduction of the personal computer was the spreadsheet. The spreadsheet provides a means for simple data input and display without compromising functionality. In other words, it provides an easy to use and friendly interface and does not eliminate the ability to perform complex calculations. A spreadsheet is a program that provides a large grid of cells into which numerical data, textual data and formulas can be entered. A cell containing a data item causes the program to display that data. A spreadsheet permits a decision maker to calculate with accounting and financial data, the management of a product line or the administration of an account. It also permits the analysis and prototyping of systems. Both at the decision making level and at the system and procedural level, people

benefit greatly from the capabilities embedded in a spreadsheet.

In a spreadsheet, the computer's screen becomes a display window. The user can scroll this window vertically and horizontally to look at any part of the sheet. The spreadsheet is organized as a grid of rows labeled as 1, 2, 3... and columns labeled as 1, 2, 3.... At each intersection of a row and column there is a cell with a coordinate identifier such as R1C1, R3C4, R10C2 and so on. The total number of rows and columns that a spreadsheet may have depends on the design of the spreadsheet and the amount of resources available. The screen usually consists of two basic areas: the control panel and the window. The data window shows the displayed cells. The control panel includes an input line to input commands and data and a prompt line to display various interactive messages.

One difference between a paper spreadsheet and its electronic counterpart is size - that is, the number of columns and rows the spreadsheet can contain. Whereas a paper spreadsheet's size is limited by the size of the paper, an electronic spreadsheet has much larger size. For example, most electronic spreadsheets can contain thousands of rows and columns. Of course, the entire spreadsheet can not fit onto a computer screen at once, so at any given time, users see only a piece of the entire form. This piece is called a window, and users can position it on the form anywhere they want.

When working with a large size spreadsheet, it might be helpful to view two parts of the spreadsheet at the same time. For example, you might want to compare the total result with one or several of the spreadsheet's components in the same window, but it is impossible to display all information in the same window for a huge size spreadsheet. Therefore, some electronic spreadsheet programs provide a **split screen** feature in which each part of the computer screen is a window displaying a different part of the large size spreadsheet. In Figure 2.1, the upper screen shows the part of the spreadsheet that fits on one screen. The totals, however, are located in row 16, which does not fit on the this

screen. Therefore, the lower screen shows how the split screen feature works. In one part of the screen, the user sees rows 1 through 3. In the other part of the screen, the user sees rows 14 through 16. Using a **split screen**, the user can combine the display of several parts of a huge size spreadsheet in the same window.

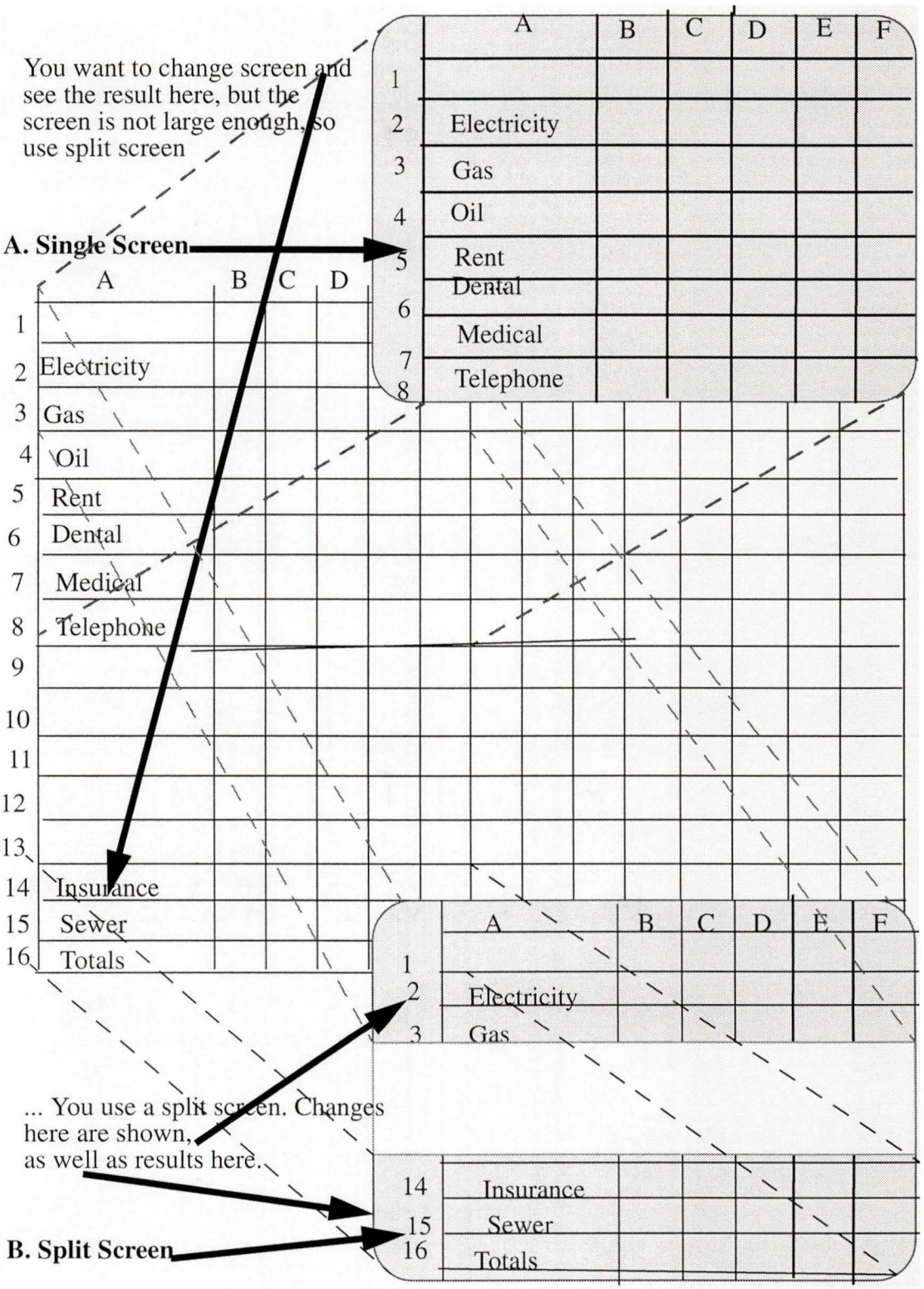

Figure 2.1 Single screen and split screen

In numerical summaries, such as budgets and expense lists, an electronic spreadsheet computes totals, averages, and other values. These applications are simple and straightforward. They simply record financial facts. Calculations allow various subtotals and totals to be quickly determined and displayed. Generally, a spreadsheet supports intuitive and visual displays and can be used for analysis and calculations.

In general, a spreadsheet consists of a window containing a two dimensional matrix of cells in which individual data elements are displayed. It is difficult to visualize multidimensional information using a two-dimensional window. In order to solve this problem, a multidimensional spreadsheet is necessary. A multidimensional spreadsheet can assist in visualizing multidimensional data [Sta93]. A multidimensional spreadsheet provides a two dimensional view into the complex data. Two steps are necessary in selecting a two dimensional view of multidimensional data. These steps are as follows:

1. Specify any two dimensions to map to the X and Y axis.
2. Fix the other dimensional position values.

Clearly a multidimensional spreadsheet only provides a two-dimensional window. However, displayed information can have a multidimensional relation by specifying two dimensions and fixing all other dimensional position values. Within a multidimensional spreadsheet window, users see all the mapped X and Y axis dimensional positions with a fixed position value for all the other dimensions. A multidimensional spreadsheet-style browser provides a spreadsheet-style window to display and perform calculations on multidimensional data.

Our implementation of a multidimensional browser uses a header file to describe the warehouse dimensional structure. Using this, our browser has vital information about the warehouse. We can then choose any two dimensions from all available dimensions.

The two selected dimensions are mapped to the vertical and horizontal dimensions. The other dimensional positions are fixed. Based on these settings, we can generate a spreadsheet-style window to view data. The header file and user selection enable us to view any multidimensional data from different viewpoints. By selecting different viewpoints, we can analyze and perform calculations on the multidimensional data. Our browser will not only be used with multidimensional output data of GLU programs, but also with any multidimensional data warehouse of similar structure. In addition, our browser supports some features of OLAP. The basic concepts of OLAP are introduced in the following section.

2.3 OLAP concepts

The concept of OLAP is defined in [E. F. Codd, S, B, Codd, OLAP *(On-Line Analytical Processing) with TM/1,* 1994.] as follows.

> OLAP is the name given to the dynamic analysis required to create, manipulate, animate and synthesize information from different data analysis models. OLAP includes the ability to discern new or unanticipated relationships between variables, the ability to identify the parameters necessary to handle large amounts of data, to create an unlimited number of dimensions, to specify cross-dimensional conditions and expressions and to analyze data according to these multiple dimensions. Multidimensional analysis is one of the major characteristics of OLAP.

In general, OLAP is used to effectively generate additional information on operational data. In many cases, OLAP provides dramatic analysis and summaries from different viewpoints. OLAP allows numerous, speculative "what-if" and/or "why" data model scenarios. Within these scenarios, the values of key variables or parameters are changed, often repeatedly, to reflect potential variances of such things as supply, production, economic, sales, marketing, costs etc. In OLAP, information is synthesized through animation of the data model. Usually, a data model contains the consolidation of data according to more than one data consolidation path or dimension. Results of on-line ana-

lytical processing are normally displayed on terminals, but they may also have to be recorded in other formats for further reference. When these results are stored in a data warehouse, it is necessary to be sure that speculative data is not confused with data that represents the actual state of the enterprise.

In addition, OLAP servers should support common analytical operations. These operations should include data consolidation, drill-down, and "slicing and dicing". These terms will be explored in more detail in the following paragraphs.

Data consolidation is described in [E. F. Codd, S. B. Codd, C. T. Salley, *Providing OLAP (On-line Analytical Processing) to User-Analysts: An IT Mandate*, 1993.] as follows:

> Data consolidation is the process of synthesizing pieces of information into single blocks of essential knowledge. The highest level in a data consolidation path is referred to as that data's dimension. Data consolidation path is a series of consolidation levels or steps defined in terms of multilevel parameters.

In OLAP, a given data dimension means a specific perspective of the data included in its associated consolidation path. There are typically a number of different dimensions. A given pool of data can be analyzed from these different dimensions. This specific perspective (or Multi-dimensional Conceptual View) appears to be the way most business persons naturally view their enterprise. Each of these perspectives can be considered to be a complementary data dimension. Simultaneous analysis of multiple data dimensions is referred to as multi-dimensional data analysis.

Sets of data can be consolidated in a vast number of different ways. Each data consolidation path reflects the perspective and intent of its creator. It is possible that some people are interested in some facts at some particular points in time, while other people may be interested in other facts at other points. Therefore, simultaneous analysis of multiple data dimensions (i.e. multi-dimensional data analysis) is necessary. Data consolidation paths consist of series of consolidation levels or steps that are defined in terms of

multi-level parameters. These parameters apply to values from any variable where each successive level represents a higher degree of data consolidation. Figure 2.2 explains an example which is originally written in [CC94]. In this example, "business enterprise" could serve as a consolidation path for the variable "vendor". "Business enterprise" might include as a consolidation path the levels "division", "department", "product line", and vendor". Data at a consolidation point is calculated by applying an aggregate operation (sum, max) to the data at lower points.

Once data has been consolidated according to one or more consolidation paths, drilling down and rolling up become feasible. Drilling down refers to the movement from higher to lower levels of consolidation for detailed information. Conversely, rolling up refers to the movement from lower to higher levels of consolidation for statistical information. Our browser provides only a limited form of drill down/ roll up.

"Slicing and dicing" refers to the ability to look at the data warehouse from different viewpoints. Figure 2.3 gives an example originally written in [Men93] to explain what "slicing and dicing" is. In this example, a cube can be seen as a multidimensional model. Product managers may focus on one product across many time periods and markets. Financial managers may focus on the current and previous time period for all products across some markets. Strategic planners may focus on a subset of the corporation's data, such as the current and next quarters for an innovative product being sold only in the West. Users are able to develop views of the data warehouse without having to know the internal structure of the warehouse. Viewpoints are selected in a manner similar to the way a child rotates an alphabet block to see a different letter. "Slicing and dicing" lends itself well to many applications but is particularly useful along a time dimension in order to analyze trends and find patterns.

Often, the cube is used to visualize the multidimensional model. However, data warehouses are by no means limited to three dimensions. Comprehension of more than

three dimensions is very difficult. Users generally need only examine a two-dimensional slice and fix the other dimensional positions. With this in mind, the cube model is sufficient. Our browser is based on this idea that a two dimensional portion of a multidimensional warehouse can be presented to the user and produce valuable insights.

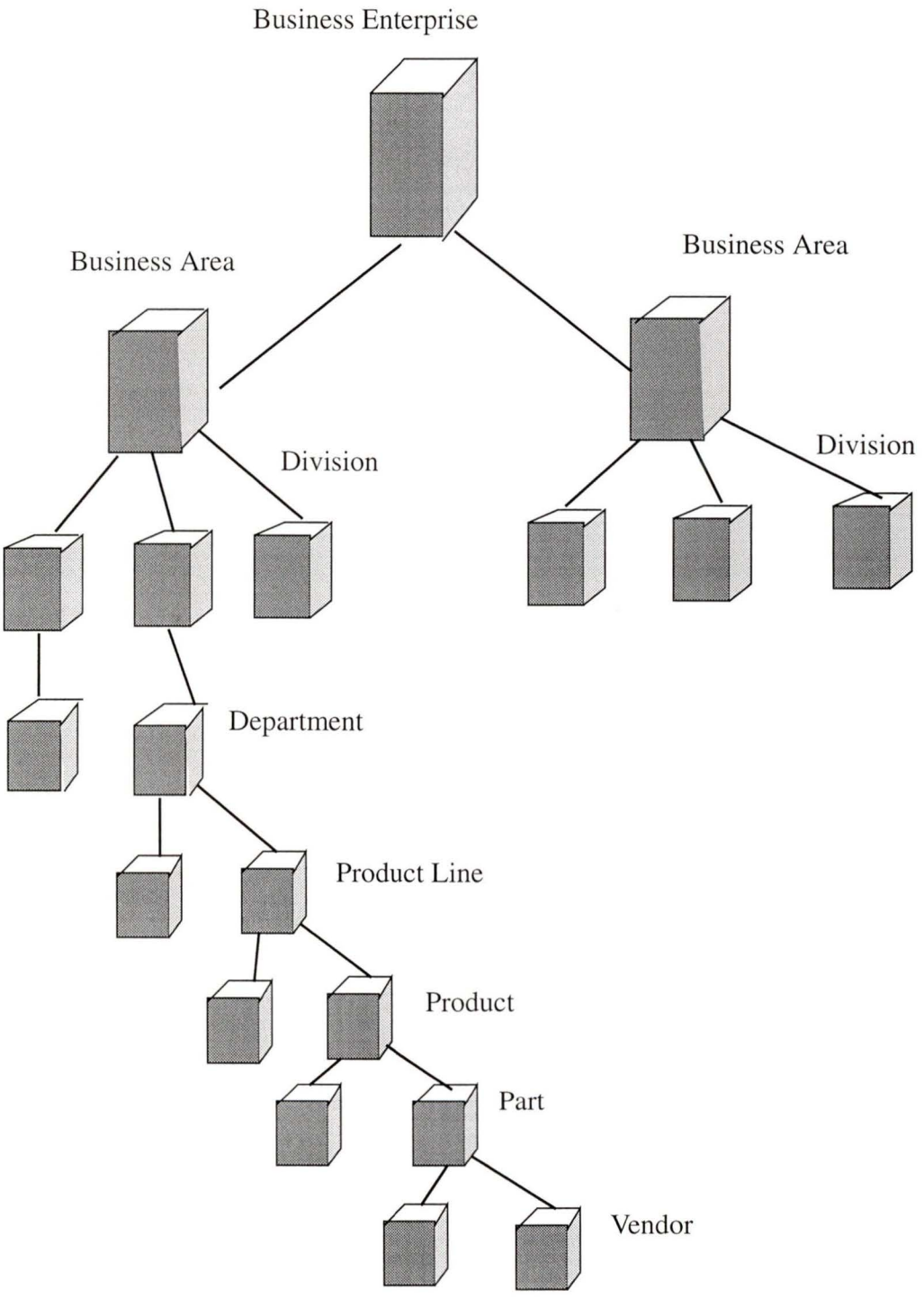

Figure 2.2 An example of Data Consolidation Paths

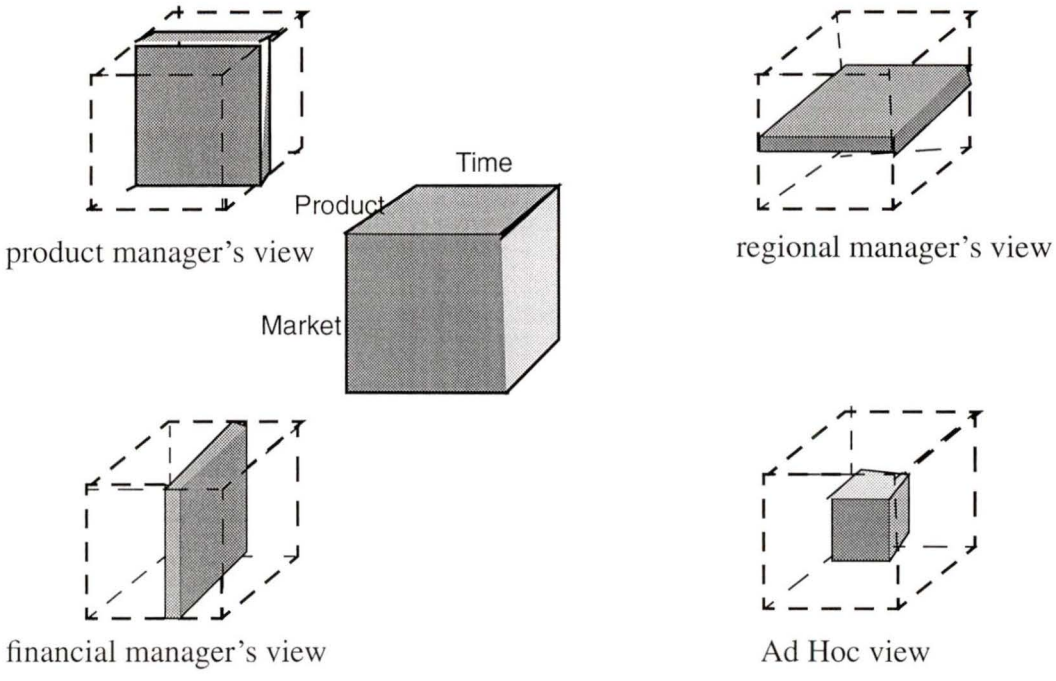

Figure 2.3 Sample views supported by multidimensional data

Chapter 3

Design & implementation of a spreadsheet-style browser

3.1 Overview

GLU is a combination of LUCID and the language C. C functions are used as building blocks for LUCID based applications. LUCID in its most recent form is a multidimensional programming language with the characteristics of a dataflow language. Because dataflow languages output a series of dataflow values generated by multidimensional programming, GLU users do not know what the meaning of a series of values is and the relations among these dataflow values. With this in mind, users struggle with the output. Therefore, in order to solve this problem, we designed and implemented a multidimensional spreadsheet-style browser for analyzing output data sets from GLU programs.

In order to reach this goal, several steps are needed in this thesis. The first one is to extract multidimensional data from GLU programs, and organize this output data into a multidimensional warehouse. The second one is to get specific data values in this multidimensional warehouse. The third one is to visually display these multidimensional data sets. The final one is to calculate & analyze these multidimensional data sets. In Figure 3.1, the process for browsing output data of GLU program is depicted.

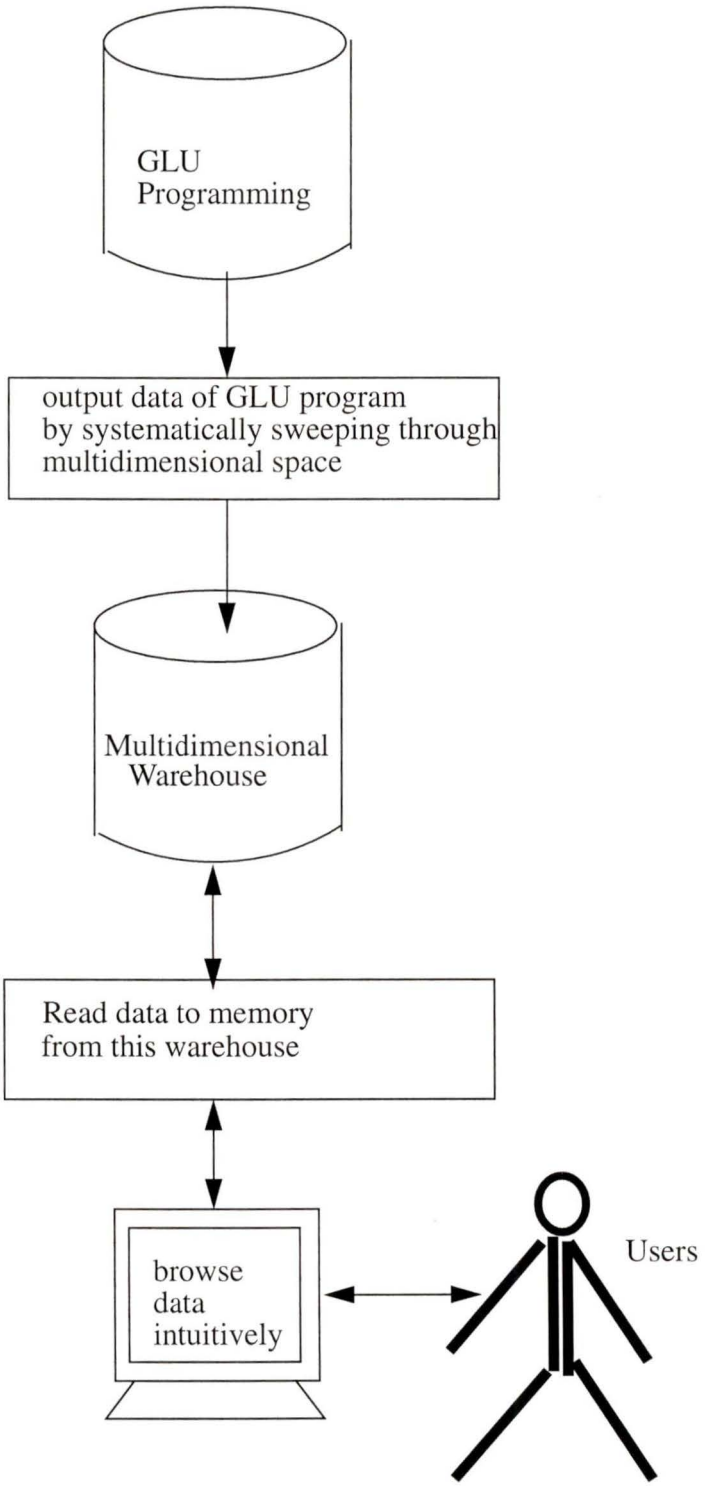

Figure 3.1 The process for browsing output data of a GLU program

How to navigate dimensional positions for multidimensional output data in GLU, how to store these multidimensional data sets in a warehouse in order, how to access this warehouse data and how to implement a browser will be explained in later sections.

In this thesis, the browser can visually display the meaning of a series of values and assist in analyzing these multidimensional data sets. This browser can also give output values meaning while supporting data consolidation, drill-down, and "slicing and dicing" etc. In the later chapters, we will explore data consolidation, drill-down, and "slicing and dicing". As expected, visualizing more than three dimensions is difficult, but fortunately users do not have to. Users generally need examine only a two-dimensional slice on the screen at one time, without knowledge of the underlying data warehouse structure. Therefore, our browser is a multidimensional spreadsheet-style browser to view multidimensional data by choosing any two dimensions. In addition, this browser can be seen as a general purpose multidimensional browser with some characteristics of OLAP. It can browse any multidimensional warehouse with similar structure.

3.2 Approaches to collecting output data

We have devised two approaches to collecting output data from GLU programs. This section will explore these approaches and the differences between them.

The first method embeds the invocation of the browser directly in the GLU code. The GLU programmer can directly call a BROWSER function inside a **where** clause. Generally, this approach is good for navigating "local" dimensional positions of a GLU program. From GLU, programmers could call the BROWSER function like this:

```
Local type browser(int1, int2, ..., intn, type )
.......................
result = browser(dim1, dim2, ...dimn, value);
value = /* compute formula*/;


/*
```

```
*result = name of output variable for multidimensional data sets
*n = number of dimensions
*type = a C language object type for value;
*value = computation value of the output variable;
*browser = a C language function;
*/
```

Figure 3.2 A description of navigating dimensional positions inside **where** clause

An example in Figure 1.2 already gave a detailed explanation using this approach, so we do not need to explain more about this approach here. The advantage of this approach is to let the GLU programer directly call a function browser to navigate all dimensional positions with a simple modification of the GLU program. In other words, we do not need to modify any of the computations of the GLU program.

Generally the BROWSER function call is applied to the variable of the GLU program which is expected to navigate its dimensional positions. The browser function itself should be run following the GLU program. By placing the BROWSER function call in the GLU program, the users can not only immediately display the data but also save the data generated by the GLU program execution. In this thesis, this approach is used only to collect the output data of a GLU program, not to display it. The GLU program generates and stores data in a data warehouse by navigating dimensional data and these data can be used later by a general purpose BROWSER program. Figure 3.1 shows the general process for browsing the output data of GLU programs using this approach.

The second approach which we found is to call a GLU program from a C main() routine. In other words, in main() of C language, a function calls the GLU program to browse the output of the GLU program. The coordinates for each of the "global" dimensions are specified in the function as arguments. In a C main() routine, a GLU program can be called like this:

```
main() /* This is a C main function */
{
.................
for (dim1 =0 to len(dim-1))
 for (dim2=0 to len(dim2-1))
   ..................................
    for (dimn = 0 to len(dimn-1))
     printf("[%d, %d, ... %d] = %f \n", dim1, dim2, ... dimn,
            trace(0, dim1, dim2, ... dimn)
            );
.................;
}

 // GLU program part like this form:
dimension dim1, dim2, ... dimn;
trace(x) {type trace(int x, int dim1, int dim2, ... int dimn)} = ..
.........
```

Figure 3.3 A description of navigation using a C main() routine

Next, we use the example of the GLU program from figure1.1 to further explain this idea. We can modify this GLU program like this:

glu.g

```
dimension a, b, c;
     f( x ) {int f( int x, int a, int b, int c)} =
      (#.a) + (#.b)*10 + (#.c)*100;
```

glu.c

```
#include "glu_glu.h"
#include <stdio.h>

main(argc, argv)
     int argc;
     char *argv[];

     {
```

```
int i, j, k;

   glu_init(&argc, argv);
   for (k=0; k<7; k++)
     for (j=0; j<5; j++)
       for (i=0; i<4; i++)
   {
      printf("[%d, %d, %d] = %d\n", i, j, k, f(0, i, j, k));
   }
   glu_fini();
}
```

Figure 3.4 An example of navigating GLU program by a C main() routine

After compiling and executing this program using "gc -gen glu.g glu.c", the following output data may be generated.

```
[dim. pos.] = values
[0, 0, 0] = 0
[1, 0, 0] = 1
[2, 0, 0] = 2
[3, 0, 0] = 3
[0, 1, 0] = 10
[1, 1, 0] = 11
[2, 1, 0] = 12
.....................

[3, 3, 6] = 633
[0, 4, 6] = 640
[1, 4, 6] = 641
[2, 4, 6] = 642
[3, 4, 6] = 643
```

This approach is good for navigating "global" dimensional positions. However, this approach does not trace the GLU program running. Another disadvantage of this approach is that all C call functions are synchronous which makes these functions very

inefficient because the GLU computation starts from scratch with every call. Of course, we can make them asynchronous by RPC (remote procedure call), but it is not simple.

In general, we think the first method is good for "local" dimensional positions and the second method is good for "global" dimensional positions. It may be more attractive to combine the two methods to work together for "local" and "global" dimensional position navigation. In this thesis, we just simply introduce the two approaches to verify that dimensional positions can be navigated and that we can browse these multidimensional output data with dimensional positions after collecting these data into a multidimensional warehouse. We did not further research combining the two approaches to navigate all dimensional positions in this thesis. Our main goal is to provide a way to view intuitively the multidimensional output data of GLU programs. In chapter 4, therefore, we will give an example to verify that the multidimensional output data of GLU programs can be browsed by using the first way. We choose the first approach to solve our problem since this way easily gives users a chance to analyze multidimensional data sets, almost eliminates the need to modify existing GLU code, and gives us a simple way to download GLU output data into a warehouse. In addition, the important aspect is that the first approach just follows the original GLU program running. The following is to explain further how to browse output data from a GLU program.

3.3 Implementation of the browser

3.3.1 Organizing a multidimensional warehouse

We know that the output of GLU programs is multidimensional data sets. In order to browse and analyze these data sets, a browser simply needs an understanding of the output data structure. This is the key information needed by the browser. Therefore, a GLU output data structure can be (almost) any valid C object such as chars, ints, floats,

doubles, fixed-length arrays, structures, and pointers to any of these. The first task is to implement a get_output_structure function which is used to get the output structure from the GLU program. First of all, the GLU programmer should define a structure in a file describing the GLU program output. The browser obtains the output data structure by interpreting this file. The principle of interpreting this file is similar to lexical and grammar analysis using "Lex and Yacc" in Unix. We do not go into much detail on this topic as it is not the thrust of this thesis.

All output data sets of a GLU program will be stored in a multidimensional warehouse. There are two parts to this multidimensional warehouse. One is the warehouse header which includes some format information such as the warehouse size, cell type, and consolidate path etc. The other part is the actual data block containing all data values or data. See figure 3.5.

In figure 3.5, a warehouse header is used to determine which specific cell is requested. This allows for the data to be accessed directly. The warehouse header consists of the number of cells, the number of dimensions, dimension name, coordinate name, and consolidate path etc. To access data, the header is used to determine which specific cell is requested. This allows the data to be directly accessed in a warehouse file.

Because this warehouse is the combination of a header plus data, it is possible that multiple warehouses can be efficiently indexed in order to access large volumes of multidimensional data.

Because UNIX files are one dimensional in nature, it is absolutely necessary to have an order to store all multidimensional data to a UNIX file as a multidimensional warehouse. In fact, the key point is how to choose dimensional order and how to change all dimensional coordinates by order. In other words, there are two aspects for storage order. The first one is how to choose a dimensional order. That is, first change the first dimensional coordinate, then change the second dimensional coordinate; or first change the last

dimensional coordinate, then change the second last dimensional coordinate; or just randomly choose a dimension to change. The second one is how to change all dimensional coordinates by order. That is, change a dimensional coordinate in a descending, ascending or random order. In this thesis, the last dimensional coordinates will be chosen to change first, then second last dimensional coordinates will be chosen to change second until the first dimensional coordinates are chosen to change last and every dimensional coordinate will be changed in an ascending order because a multidimensional array is also stored in the same order in the C language. The following example is used to explain the storage order. See figure 3.6. There are three dimensions D1, D2, D3 whose lengths are 2, 3, 5 respectively.

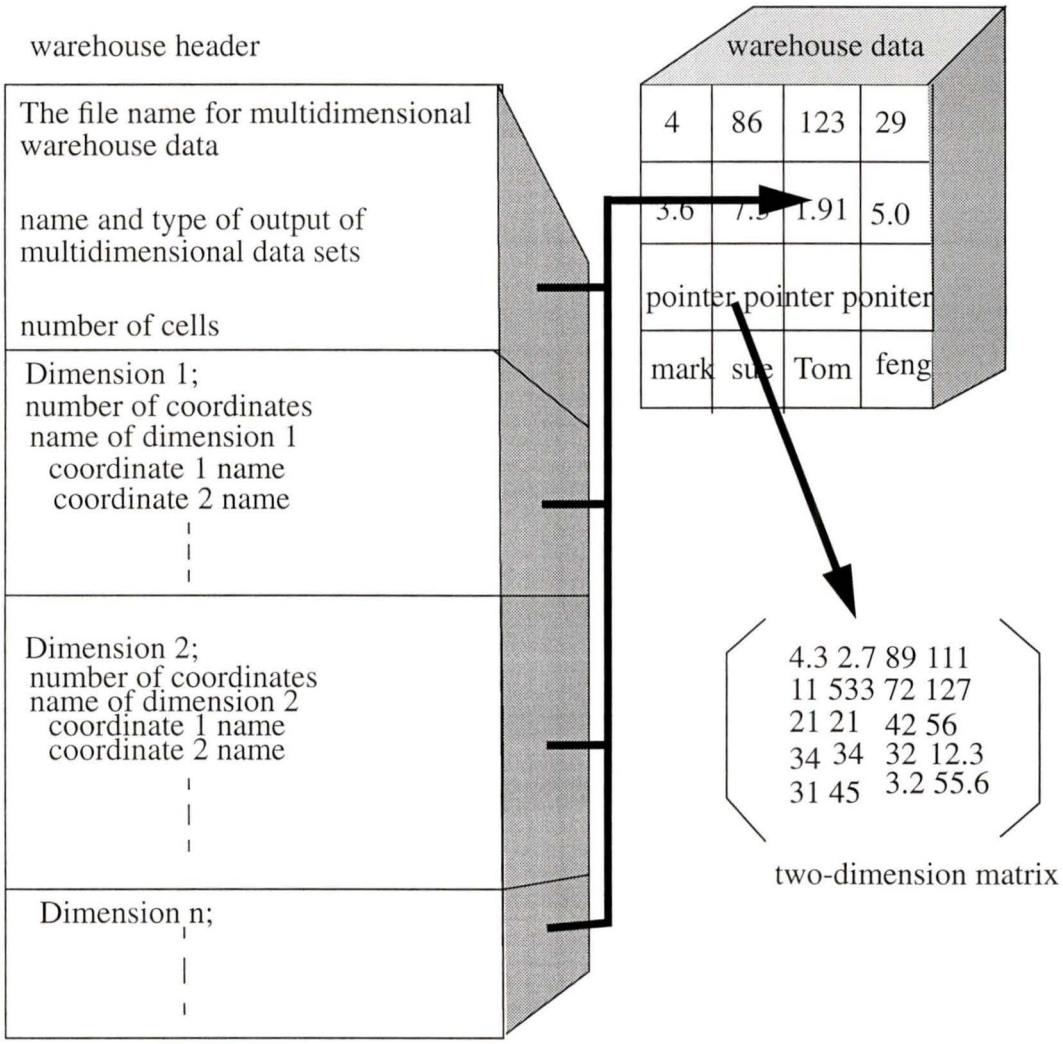

Figure 3.5 general structure of a multidimensional data warehouse

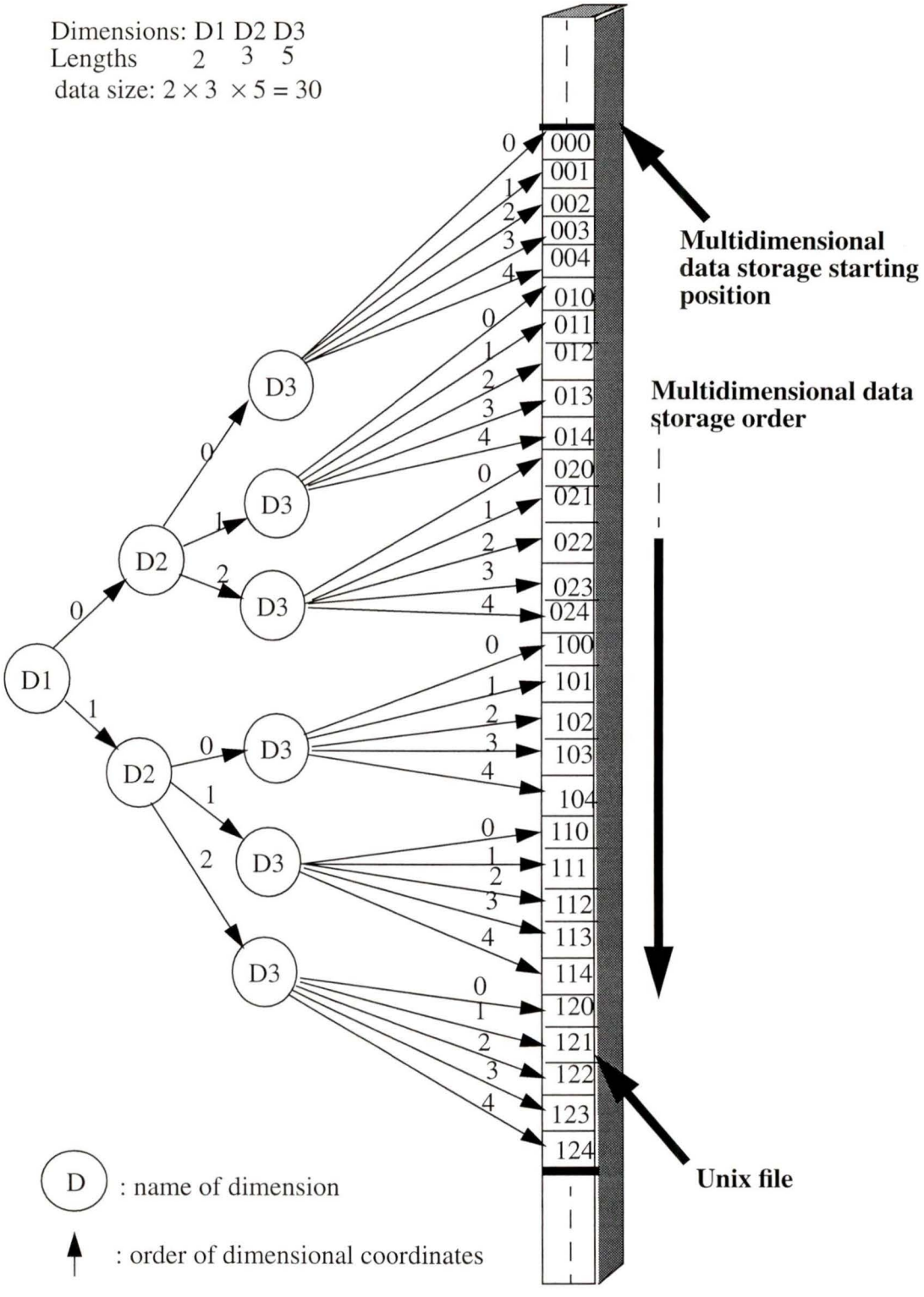

Figure 3.6 multidimensional warehouse data storage order in a UNIX file

In the previous figure 3.6 about storage order, dimensional coordinate change is propagated from the last dimension to the first, therefore, we can use the following general formula to get a specific data address:

Without loss of generality, suppose that:

There are n dimensions, D1, D2,..., Di-1, Di, Di+1,..., Dn;

Each dimensional length is L1, L2,..., Li-1, Li, Li+1,..., Ln respectively;

Each dimensional specific coordinate is S1[t], S2[t],..., Si-1[t], Si[t], Si+1[t], ..., Sn[t];

Then, total multidimensional data size in warehouse is: L1 × L2,..., × Ln;
the specific data address is:

$$\begin{aligned}\text{Addres}[t] = {} & S1[t]\times L2\times L3,...,\times Ln + S2[t]\times L3,...,\times Ln\ ),..., + Si\text{-}1[t]\times Li,...\times Ln \\ & + Si[t]\times Li+1\times,...,\times Ln + Si+1[t]\times Li+2,...\times Ln + Sn\text{-}1[t]\times Ln + Sn[t] \qquad (3.1)\end{aligned}$$

This formula also can be written as follows:

$$Address\,[t] \;=\; \sum_{i\,=\,1}^{n}\left(Si\,[t] \bullet \prod_{j\,=\,(i\,+\,1)}^{n} Lj \right)$$

when i = n, making the following equation in the above formula:

$$\prod_{j\,=\,i\,+\,1}^{n} Lj \;=\; 1$$

For this formula, a dataflow idea can be used. Multidimensional coordinates can be specified step by step. Therefore, the address is reached time by time. The formula actually denotes a stream of values. In other words, if a specific stream of multidimensional coordinates is <s1[0], S2[0],..., Sn[0]>, <s1[1], S2[1],..., Sn[1]>,..., <s1[t], S2[t],..., Sn[t]>, the formula can yield a stream of addresses, Address[0], Address[1],...,

Address[t]. Therefore, when a GLU program generates a multidimensional data set which has specific coordinates for each dimension, this formula yields the storage address based on the specific coordinates for each dimension, thus storing the output to the newly computed address. This address is used in the data retrieval phase for the browser as well as for data storage mentioned above. Later discussion will address how to get a specific data set with its dimensional coordinates and how to visualize these data sets.

3.3.2 Input of multidimensional warehouse data

In this thesis, the multidimensional warehouse is a Unix file which is stored on disk, therefore, access is expensive due to disk I/O operations. There are three approaches to reading data from disk to memory. The first one is for saving time, but spending space. In other words, all multidimensional warehouse data are read into memory. Another one is to read only needed data from disk to memory. For example, after users choose any two dimensions as the x-y axis and fixes all other positions, only the data in these two chosen dimensional all positions and other dimensional fixed positions are read into memory based on formula 3.1. The last one will not read data from disk into memory until all dimensional positions are specified, but this way is too slow to display data. In this browser implementation, because we have enough memory to store these data in a SUN workstation, and for more efficient analysis and calculation, we choose the first approach: all multidimensional warehouse data will be placed in a one-dimensional array in memory. In this array, every data access address will also be computed based on formula 3.1. Although it is feasible to use a multidimensional array to store the multidimensional data, it is actually meaningless because this multidimensional array in the C language also uses formula 3.1 to compute the address. Therefore, a one dimensional array in C is used to store our multidimensional data. The size of the multidimensional data is equal to the product of every length of all dimensions and the number

of dimensions. Knowing that the length of every dimension is not constant, the size of the multidimensional data is variable. In an effort to make efficient use of memory, dynamic memory management will be utilized for our array space. In order to get the format information of multidimensional data, an interpreter program will be implemented to process all dimensional information from the warehouse header file. This will convey format information about the warehouse size, cell types, and number of dimensions in the warehouse header file to the browser. For example, a warehouse header file exists describing the structure for every dimension of each dimensional coordinate. See figure 3.7. In this example header file, a data cell is declared as type **float**; the number of dimensions is 5; the names of our dimensions are declared as "Salemen", "Nation", "FiscalYear", "Product", and "Version" respectively. In addition, the length of every dimension and name of every dimensional coordinate are also specified.

```
//* This is a file to describe the multidimensional data format *//

DATA_TYPE="float"

NUMBER_OF_DIMENSION=5

DIMENSION_NAME="Salemen"
DIMENSION_LENGTH=7
SUB_NAME="Tom"
SUB_NAME="Mark"
SUB_NAME="Kevin"
SUB_NAME="Bill"
SUB_NAME="Chris"
SUB_NAME="Jack"
SUB_NAME="John"
```

```
DIMENSION_NAME="Nation"
DIMENSION_LENGTH=25
SUB_NAME="Canada"
SUB_NAME="U.S.A."
SUB_NAME="Mexico"
SUB_NAME="Argentina"
.....


DIMENSION_NAME="FiscalYear"
DIMENSION_LENGTH=12
SUB_NAME="January"
SUB_NAME="February"
SUB_NAME="March"
......

DIMENSION_NAME="PRODUCT"
DIMENSION_LENGTH=20
SUB_NAME="Apple"
SUB_NAME="Pear"
SUB_NAME="Peach"
......

DIMENSION_NAME="Version"
DIMENSION_LENGTH=3
SUB_NAME="Min_Budgeted"
SUB_NAME="Actual"
SUB_NAME="Max_Budgeted"
```

Figure 3.7 An example of a multidimensional warehouse header file

There are some keywords in the multidimensional warehouse header file. The interpreter program surveys format information by these keywords. These key words are:

1. DATA_TYPE
2. DIMENSION_NAME
3. DIMENSION_LENGTH
4. NUMBER_OF_DIMENSION
5. SUB_NAME

Figure 3.8 The keywords for warehouse header file

Once the following two tasks are completed, the next task is how to display multidimensional data sets.

1. Interpret multidimensional data format information for use by this browser.
2. Read multidimensional warehouse data sets and store in our array. In other words, read the multidimensional data sets from disk to memory.

3.3.3 Visual display

Difficulties in visualizing more than three dimensions simultaneously can be avoided by systematically browsing two dimensions at one time. In other words, users can choose any two dimensions and have them mapped on the browser's X-Y display. The other dimensional coordinates are fixed by the users. When users invoke the browser, the first and second dimensions in the list are, by default, mapped to the browser's X-Y display. All other dimensional coordinate values default to the coordinate with index ZERO. Based on these default values, a spreadsheet-style view will be displayed with its origin in the upper left of the window. Because of the limit of the size of the window, the view of 5 rows and 5 columns was chosen, therefore, at most there are only 25 cells showing in this spreadsheet-style view at one time. The other cells can be shown by using the scroll-bar on the X or Y axis.

The browser displays a two-dimensional view. The number of columns is equal to the length of the X dimension list while the number of rows is equal to the length of the Y dimension. Because users can choose any dimension as the X Dimension List or the Y Dimension List and the length of each dimension may not be the same, the size of the spreadsheet form may change. In other words, the variable X Dimension List or Y Dimension List dimension leads to a different size of the new spreadsheet form. Each row-column intersection generates a cell. Each cell can contain a single size of data value. Users can view very large spreadsheet forms by using the browser's horizontal and vertical scrollbar controls. In order to obviously show all dimensional list order, the browser gives three consecutive buttons for X Dimensional List, Y Dimensional List and the Other Dimensions List respectively. Any one button represents a single dimension in the dimensional list. Each dimension in these three Dimensional Lists is selected by a vertical scrollbar in three consecutive buttons. Users can click one button as choosing a dimension in these three buttons for every Dimensional List. Selecting the same dimension for both the X and Y axes will yield a one dimensional display giving very little insight into the multidimensional data. To prevent such a situation, X Dimensional List's, Y Dimensional List's and the Other Dimensions List's dimensional specification is mutually exclusive. For example, if users choose D1 as the X Dimension List's dimension, they will be unable to choose the D1 as Y Dimensional List's or Other Dimension List's dimension again. However, deadlock can result from this "mutual exclusion" restriction. For example, users have set the X-Y-Other Dimension List to dimension D1-D2-D3 and would like to change the Y Dimension List dimension D2 to D1 or D3, but D1 and D3 are already occupied by X and Other Dimension List. Deadlock avoidance can be implemented by assigning a priority to each Dimension List with X Dimension List having highest priority and Other Dimension List having the least priority. In other words, the X Dimension List can be any dimension in all dimensions. The Y Dimension List can be any dimension except what is specified by the X Dimension List. The Other Dimension

List can be any dimension except what is selected in both the Y Dimension List and Y Dimension List.

When X and Y Dimension List's dimensions are set, all other dimensional coordinates will be specified by a scale. In the browser, Other Dimension List not only sets other one dimension, but also specifies this dimension's coordinate. Therefore, to the right of the Other Dimension List, there is a horizontal scale to specify this dimension coordinate. Users can click or drag the mouse along in the scale, selecting a dimensional coordinate in the range from 0 to length of the dimension minus 1. Users can also choose any two ranges for the X or Y Dimension List's dimension. The default ranges for the X and Y Dimension List's dimensions are from 0 to (length-1). Upon completion of the selection for the X Dimension List's, Y Dimension List's dimension and Other Dimension List's dimension coordinates and the ranges for the X or Y Dimension List dimension, the browser can generate a spreadsheet-style view according to the user's wishes.

3.3.4 Analysis and Calculation

Now, a spreadsheet-style browser has been generated. Users can use the browser to analyze and calculate. In this thesis, there are mainly three parts to be discussed. They are "slicing and dicing", "comparison and combination" and "calculation information".

The first one is "slicing and dicing" which is an important feature for an OLAP system. In chapter 2.3, we already discussed its concept and function and do not need more explanation here. A major benefit of "slicing and dicing" is to give the users the ability to analyze trends and find patterns from different viewpoints. This browser allows an unlimited number of dimensions. When users choose any two dimensions as the X Dimension List and Y Dimension List and fix the other dimension coordinate values in the Other Dimension List, the browser can generate any slice they like. When users specify all coordinate values for every dimension, the browser can show any dice they wish. Slicing and dicing can be a powerful tool in the visualization of complex multidimensional data.

The second one is "comparison and combination". Because the length of a dimension may be very large, the size of the spreadsheet form may become very huge if users choose this dimension as X Dimension List's or Y Dimension List's dimension. Usually a browser just displays a part of the entire form using X and Y Dimension List consecutive dimensional coordinates. Therefore, it is difficult for users to display and compare a part of the spreadsheet form with another part of the spreadsheet form at the same view when the difference of two part coordinates is very large in the form. In order to solve this problem, any two ranges can be selected and displayed simultaneously in a split screen format. This kind of spreadsheet is called a combinational view. Each part of this view is useful for users and can be compared and combined as the users wish. In chapter 4, an example will be given to explain the details of comparison and combination functions.

The last important feature of this browser is the ability to do calculation on the data. Some calculation information can be performed on numerical data contained in all cells. Four calculation functions are available for the X or Y Dimension List. These are:

1. MIN_VALUE calculates minimum value for a selected row or column.
2. MAX_VALUE calculates maximum value for a selected row or column.
3. AVE_VALUE calculates average value for a selected row or column.
4. SUM_VALUE calculates sum value for a selected row or column.

For a combinational and comparative browser view, users can also use these calculation functions for analysis and comparison. In chapter 4, an example will be given to explain the details of these calculation functions.

This browser is implemented using C language and X-windows running on a UNIX based system. The C language was selected due to its portability and compatibility with GLU.

We think that this browser can not only be used as an interface for displaying and

analyzing output data from GLU, but also it can browse any multidimensional warehouse with a similar structure. In chapter 4, we will give an example of a GLU program to verify and show how to view multidimensional output data of this GLU program by our browser.

Chapter 4

Verification and use of the browser

4.1 Characteristics of the browser

Following are some of the characteristics of this spreadsheet-style browser:

1. The browser provides a graphical view of multidimensional data sets and has some calculation ability.
2. Users can choose for further investigation any two dimensions from dimension list for X and Y Dimensions. All other dimensions are fixed at a specific coordinate using the Other Dimension List. After these operations, the browser can generate a spreadsheet view according to the user's wish.
3. The browser can compare and combine any two sections of a very large multidimensional data set at one time.
4. The browser has a user friendly interface for displaying and analyzing any multidimensional data warehouse with similar structure.

4.2 Verification and use of the browser

4.2.1 A GLU program example

In order to verify that we can browse the output data of a GLU program. we give the following example of a GLU program and a browser function to collect its multidimensional data.

verify.g:

```
test.local int browser(int, int, int, int, int, float);

((((sum@.d5 d5_pos)@.d4 d4_pos)@.d3 d3_pos)@.d2 d2_pos)@.d1 d1_pos
where
  dimension d1, d2, d3, d4, d5;
  d5_pos = #.time % 3;
  d4_pos = (#.time / 3 ) % 20;
  d3_pos = (#.time / 60) % 12;
  d2_pos = (#.time / 720 ) % 25;
  d1_unbound = #.time / 18000;
  d1_pos = if (d1_unbound < 7) then d1_unbound else eod fi;
  sum = browser(g1, g2, g3, g4, g5, x);
  g1 = #.d1;
  g2 = #.d2;
  g3 = #.d3;
  g4 = #.d4;
  g5 = #.d5;
  x = (#.d1)*1.01 + (#.d2)*1.02 + (#.d3)*1.01 + (#.d4)*1.02
      + (#.d5)*1.03;
end
```

verify.c

```
#include "verify_glu.h"
#include <stdio.h>

int  browser(int d1, int d2, int d3, int d4, int d5, float ss)
{
   FILE *fp;

      fp = fopen("verify_file", "a+");
      fprintf(fp, "%f ", ss);
      fclose( fp );
      return(ss);
}
```

Figure 4.1 An example to trace dimensional positions in GLU

After compiling and running this program, the output data of this GLU program can be collected into a warehouse, then the GLU programer needs to create a warehouse header file, data.header, to describe this warehouse data format (which is the same as in Figure 3.7). In this header file, the cell type is float, the number of dimensions is 5, and the dimensional names are "Salemen", "Nation", "Fiscalyear", "Product", and "Version" respectively. The first dimension (Salemen) has a length of 7; the second dimension (Nation) has a length of 25; the third dimension (Fiscalyear) has a length of 12; the forth dimension (Product) has a length of 20; the final dimension (Version) has a length of 10. The total size should be 7×25×12×20×3 = 126000. Every cell represents a float "Profit" value based on these five dimensional positions. In this sample program, we define the "profit" computation as following:

1. The first dimension "Salemen" increase profit by 1% as its position increases by one.
2. The second dimension "Nation" increase profit by 2% as its position increases by one.
3. The third dimension "Fiscalyear" increase profit by 1% as its position increases by one.
4. The fourth dimension "Product" increase profit by 2% as its position increases by one.
5. The first dimension "Version" increase profit by 3% as its position increases by one.

Now we can run the command "browser data.header data.warehouse" to view this multidimensional data. By default in this browser, setting the X Dimension List's dimension to "Salemen" and Y Dimension List's dimension to "Nation", setting the other dimensional coordinates ("Fiscalyear", "Product", and "Version") to ZERO, and setting the X and Y ranges [0, length-1] and [-1, -1], the browser generates the following default status view in Figure 4.2.

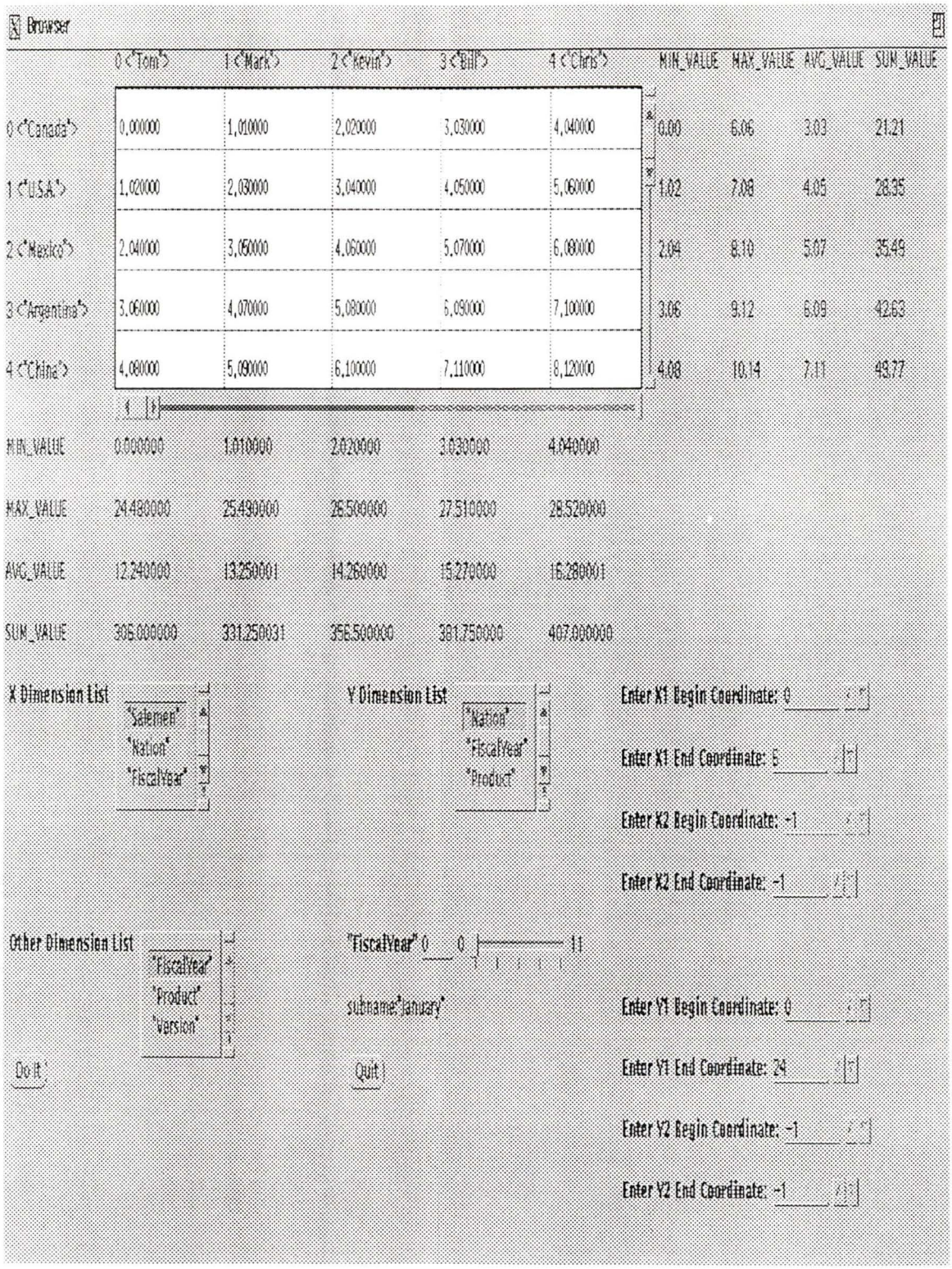

Figure 4.2 A example of default status

4.2.2 Four basic operations

After invoking the browser, users can display and calculate any multidimensional data sets they need. There are four basic operations for using the browser. These operations are how to use a scrollbar in the spreadsheet-style view to scan different dimensional coordinates for the X or Y Dimension List, how to use buttons to specify one dimension in the X, Y or Other Dimension List, how to fix the coordinate for the specific Other Dimension List dimension using a sliding scale, and finally how to use arrows to choose any two ranges for X or Y dimension. See figure 4.2.

The first operation is how to use scrollbars in the spreadsheet-style view to display and calculate multidimensional data information by scanning the X or Y dimensional coordinates. There are two scrollbars, Vertical and Horizontal. Both work on the same principle, therefore, only the Vertical scrollbar is discussed here. The Vertical scrollbar is used to scan Y dimension coordinates. The number of the dimensional coordinates is equal to the length of the Y dimension. In figure 4.2, on the right of the spreadsheet-style view is a vertical track which is called the Vertical scrollbar. In the Vertical scrollbar, there are two scrollbar arrows (up, down) and one small box. Users can scan the Y dimension coordinates by clicking the Vertical scrollbar arrow or by dragging with the small box. When users click the up scrollbar arrow, the Vertical scrollbar will move up and the spreadsheet-style view will reach the previous coordinate in the Y dimension. When users click the down scrollbar arrow, the Vertical scrollbar arrow will go down and the spreadsheet-style view will reach the next coordinate down in the Y dimension. When the Vertical scrollbar arrow hits the track top, the coordinate in the Y dimension reaches the first coordinate (0) and when the Vertical scrollbar arrow hits the track bottom, the coordinate in the Y dimension reaches the last coordinate (length-1). When users move the Vertical scrollbar, all the multidimensional data information and the corresponding calculation information will also move according to their coordinates in the Y dimension. The Horizontal scrollbar is for X and users can move it to the left or right to

display different multidimensional information based on the X dimensional coordinates.

The second operation is how to use selection lists to specify the X, Y and Other dimension. Each button shows a small rectangle. A selection list exists for the X, Y and Other Dimension List respectively. Each selection list functions on the same principle. Each selection list is made from three buttons and it only shows three consecutive dimension names at one time. Each button contains one dimensional name. A vertical scrollbar is used to traverse the entire list of dimensional name. A selection is made by clicking on the name, thus making the selection button become shaded. In figure 4.2, below the spreadsheet-style view are three selection lists for the X, Y and Other Dimension List respectively.

The third operation is how to use the scale to fix the coordinate for a specific dimension. In figure 4.2, on the right of the Other Dimension List is a scale. The scale is a horizontal track and a slider. The length of the scale is equal to the number of dimensional coordinates in the Other Dimension List. Users first choose one dimension in the Other Dimension List, then they can use the scale to choose a coordinate for this dimension. When users click or drag a slider in the scale, users can get different coordinates for this dimension. When users drag the slider to the right in the scale, the dimensional index becomes larger. As users drag the slider to the left in the scale, the dimensional index becomes smaller. If users want to change the Other Dimension List's dimension, they simply select the new dimensional name in the Other Dimension List and then use the scale to specify a new coordinate.

The final operation is how to use arrows to choose any two ranges for the X or Y dimensions. On the right of the scale is a range selection for the X and Y dimension. See figure 4.2. The upper four lines are for the X dimensional range selection; the lower four lines are for the Y dimensional range. Users can not only use two arrows (up, down) but also enter a number to specify a range for the X and Y dimension. Users can choose any

two ranges in the X or Y dimension and display multidimensional information in two ranges together.

When the user completes all selections on the screen, the user simply presses the "Do it" button to generate a new view. When the user wants to quit the browser, they can press the "Quit" button to exit.

4.2.3 Three basic functions

There are three basic functions in the browser. The first one is "slicing and dicing"; the second one is comparison and combination between separate regions of the data; the last one is to display some statistical data. Some examples will be given to explain these three functions. The multidimensional data warehouse header file format information of these examples can be found in figure 3.7.

The first function is "slicing and dicing". For example, if users would like to display and calculate a slice for which the X Dimension List dimension is "Fiscalyear" and the Y Dimension List dimension is "Product", they can simply make selections using horizontal and vertical scrollbars for the X and Y dimensions as described in a previous section. Once having chosen two dimensions "Fiscal" and "Product", users will specify all coordinates in the Other Dimension List for the dimensions "Salemen", "Nation", and "Version". If users specify the dimension "Salemen" index as 2, (i.e. sub_name: "Kevin"), the dimension "Nation" index as 23, (i.e. sub_name: "Italy"), and the dimension "Version" as 0, (i.e. sub_name: "Min_Budgeted"), they can generate a spreadsheet-style view by pressing the "Do it" button. Figure 4.3 gives an example of "slicing and dicing". The above process can be repeated to generate any new view users wish by "slicing and dicing". When users specify all coordinates for all dimensions, they can get a single dice. In fact, each cell in the spreadsheet-view is a dice. The above example also can be used for explaining "dice". In figure 4.3, when users use the Horizontal scrollbar to display one coordinate as 5 ("June") in the X dimension "fiscalyear" and the Vertical scrollbar to dis-

play one coordinate as 12 ("Computer") in the Y dimension "Product", the browser displays a cell whose value is 42.77. Used this way, users can get any dice in a multidimensional data warehouse. The "slicing and dicing" is the basis of multidimensional view concept.

The second function is how to use "comparison and combination". This browser supports the comparison of one part with another part of the spreadsheet form as well as combinational display of these two parts. An example can be used to explain this function. Start with figure 4.3, but assume the users do not want to show the entire X dimension "Fiscalyear" coordinates and the entire Y dimension "Product" coordinates; they only want to view the first Quarter and the third Quarter (i.e. coordinates are from 0 to 2 and from 7 to 9) in "Fiscalyear" dimension, and the fruit and bookstore products (i.e. coordinates are from 0 to 4 and from 17 to 19) in the "Product" dimension. After users choose these ranges, the browser can generate a comparison and combinational view. See figure 4.4. The multidimensional data warehouse header file format information of this example also can be found in figure 3.7.

The third function is calculation. The browser supports real-time calculations dependent on what is displayed in the spreadsheet-style view. Both "slicing and dicing" and "comparison and combination" examples can be used to explain the calculations. See figure 4.3 and figure 4.4. In figure 4.3, users get all dimension coordinate calculations. In figure 4.4, users get two range calculations. When the users move the Horizontal Scrollbar to left or right for the X Dimension List dimension, or the Vertical Scrollbar to up or down for the Y Dimension List dimension, all multidimensional data information will change in spreadsheet-style view, and thus all the corresponding calculated information will also change.

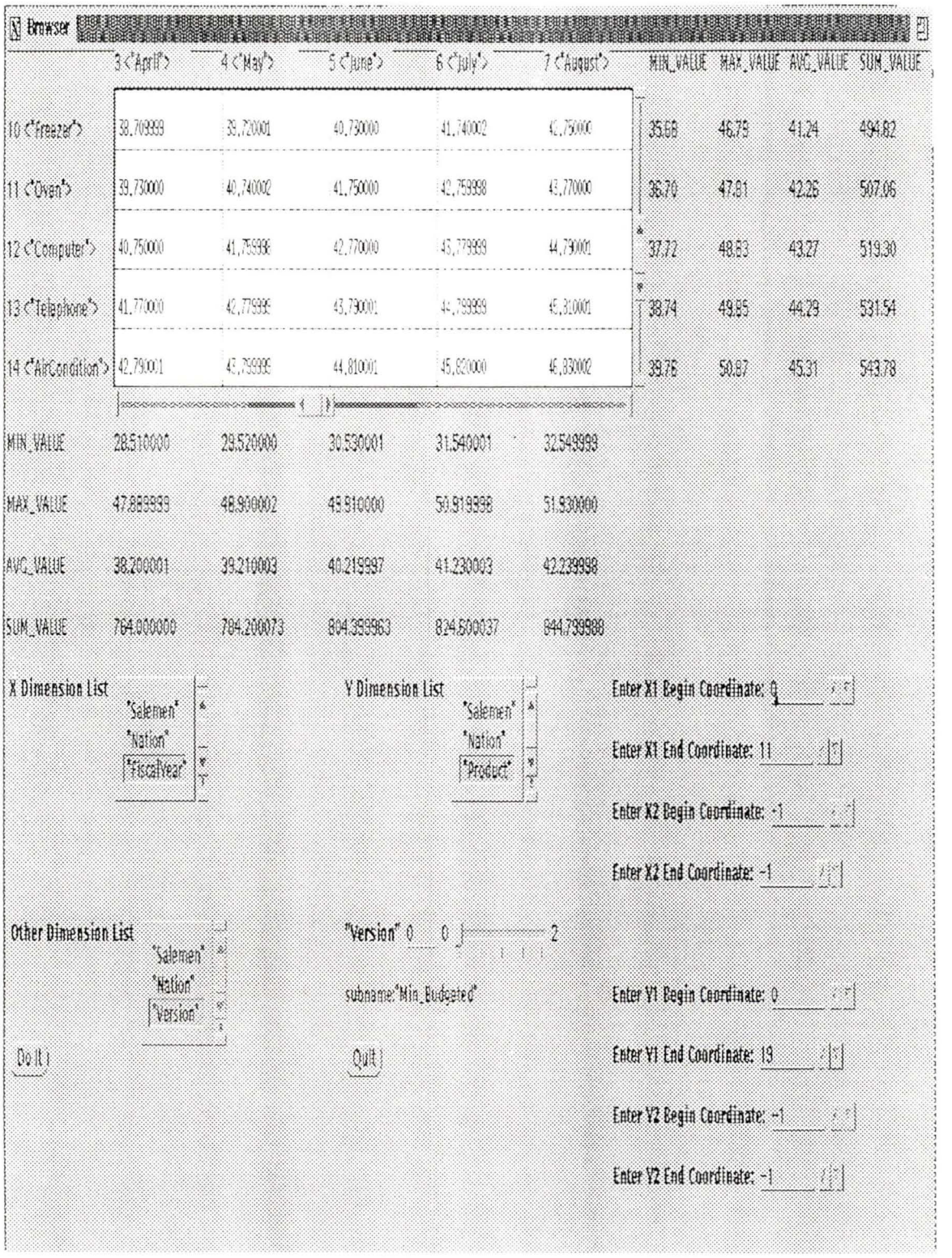

Figure 4.3 An example of "slicing and dicing"

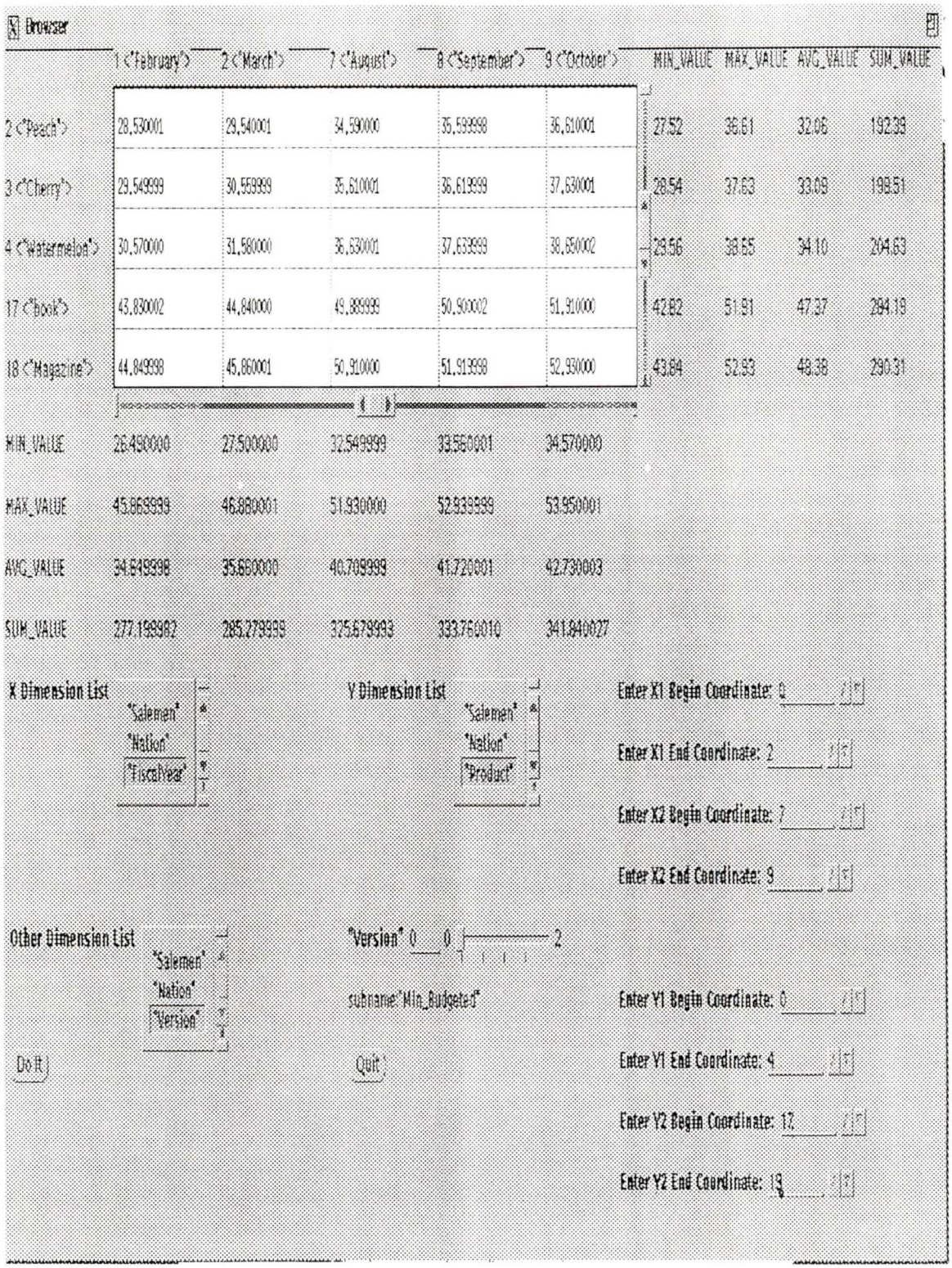

Figure 4.4 An example of "comparison and combination"

4.3 Using the browser for other multidimensional data warehouses

It is feasible that users can use this browser to analyze any multidimensional programming output, multidimensional warehouse and multidimensional data sets. The following criteria must be met in order to do so:

1. all multidimensional programming output data sets are stored in a multidimensional warehouse after evaluating all multidimensional output data sets with a multidimensional program.
2. In the multidimensional warehouse, there are two parts. One is the multidimensional warehouse header file; the other is the true multidimensional data sets. The header file must contain multidimensional format information.
3. In the multidimensional warehouse data, all data are stored by changing dimensional order from last to first dimension and changing one dimensional coordinates by ascending order. In other word, all data addresses are computed by the formula(3.1).

If the multidimensional warehouse satisfies the above three conditions, users can use this browser to analyze and calculate on their multidimensional data warehouse. The browser functions will remain the same. Due to the fact that this browser is separate from the program used to generate the multidimensional data, it is not constrained to be used for one multidimensional program. Users can browse data sets generated by a multitude of sources.

In summary, the purpose of this chapter is to verify that multidimensional output data from GLU programs can be intuitively displayed and show readers how to use this browser.

Chapter 5

Evaluating the browser using OLAP product rules

5.1 Evaluating the browser

There are twelve rules [E. F. Codd, S. B. Codd, C. T. Salley, *"Beyond Decision Support,"* page 87, Computerworld, July 26, 1993.] as follows for evaluating OLAP products;

1. Multi-dimensional Conceptual View
2. Transparency
3. Accessibility
4. Consistent Reporting Performance
5. Client-Server Architecture
6. Generic Dimensionality
7. Dynamic Sparse Matrix Handling
8. Multi-user Support
9. Unrestricted Cross-dimensional Operations
10. Intuitive Data Manipulation
11. Flexible Reporting
12. Unlimited Dimension and Aggregation Levels

This browser satisfies in part some of these rules, and we just explain these parts.

There are two paragraphs for each evaluating rule in the following. The first paragraph explains the concept which this rule is based on [CCS93a] and the second paragraph evaluates how the browser implements the rule.

Rule 1: Multi-dimensional Conceptual View

A user-analyst's view of the enterprise's universe is multi-dimensional in nature. Accordingly, the user-analysts conceptual view of OLAP models should be multi-dimensional in nature. This multi-dimensional conceptual schema or user view facilitates model design and analysis, as well as dimensional calculations through a more intuitive analytical model. Accordingly, user-analysts are able to manipulate such multi-dimensional data models easily and intuitively than is the case with single dimensional models. For instance, the need to slice and dice, or pivot and rotate consolidation paths within a model is common.

This browser provides a multi-dimensional view to the user-analyst that is very easy to declare. This multi-dimensional view browser has dimensional calculations. The browser provides "slicing and dicing", or pivot and rotate consolidation paths which work on multiple dimensions. Using this browser, views are two-dimensional views of any dimensions obtained through the browser.

Rule 2: Transparency

Whether OLAP is or is not part of the user's customary front-end product (e.g., spreadsheet or graphics package), that fact should be transparent to the user. OLAP should be provided within the context of a true open systems architecture, allowing the analytical tool to be embedded anywhere the user-analyst desires, without adversely impacting the functionality of the host tool.

This browser simply provides a spreadsheet-style window. Machine dependencies are minimized within the browser implementation thus allowing access in a variety of

ways without affecting host tool functionality.

Rule 3: Accessibility

The OLAP tool must map its own logical schema to physical data stores, access the data and perform any conversation necessary to present a single, coherent and consistent user view. Moreover, the tool should not be concerned about where or from which type of system the physical data is coming.

This browser has its own structure and order for a multidimensional warehouse which stores all physical data. This browser can access the multidimensional data sets to present different views.

Rule 6: Generic Dimensionality

Every data dimension must be equivalent in both its structure and operational capabilities. Additional operational capabilities may be granted to selected dimensions, but since dimensions are symmetric, a given additional function may be granted to any dimension. The basic data structure, formulae, and reporting formats should not be biased toward any one data dimension.

All dimensions in our browser are based on identical "primitives" or building blocks. Manipulations and calculations which can be performed in any one dimension can be performed likewise within the context of any other dimension. Users can choose any dimension to display or perform calculation. No dimensional priorities exist within this browser implementation.

Rule 9: Unrestricted Cross-dimensional Operations

In multidimensional data analysis, all dimensions are created and treated equally. The OLAP tools should handle associated calculations among dimensions and not require the users to define what these calculations should be. The various roll-up levels within consolidation paths, due to their inherent hierarchical nature, represent in outline

form the majority of 1:1, 1:M, and dependent relations in an OLAP model or application. Accordingly, the tool itself should infer the associated calculations and not require the user-analyst to explicitly define these inherent calculations.

This browser treats all dimensions equally and allows simple independent calculations on two selected dimensions. There are no cross-dimensional operations, so it does not require the users to redefine what the calculation should be.

Rule 10: Intuitive Data Manipulation

Manipulation, such as reorienting the consolidation path or drilling down across columns or rows should be accomplished via direct action on the analytical model cells. Such manipulation should not require the use of a menu or multiple trips across the user interface. The dimensions defined in the analytical model should contain all the information the user needs to put in motion any inherent actions.

Manipulation in this browser is intuitive and does not require the use of a menu or multiple trips across the user interface. The dimensions defined in the browser include all pertinent information users need. We do not support general drill down/roll up.

Rule 11: Flexible Reporting

Using the OLAP server and tools, users can manipulate, analyze and look at data any way they want, including placing rows, columns and cells close to one another visually or arranging them in logical groupings. Reporting facilities must mirror this flexibility and present synthesized information any way the user wants to view it. Reports should reflect the model.

This browser allows a split screen format. We do not have separate report-generating facilities. This browser can also present synthesized information users want to view. Therefore, it is easy to make a reporting facility mirror this information.

Rule 12: Unlimited Dimension and Aggregation Levels

Research into the number of dimensions possibly required by analytical models indicates that as many as nineteen concurrent data dimensions may be needed. Furthermore, each of these generic dimensions must allow an essentially unlimited number of user-analyst defined aggregation levels within any given consolidation path.

In this browser, there can be unlimited dimensions, but just one aggregation level. Multiple levels are not supported.

From the evaluation above, it can be seen that this browser has several characteristics of an OLAP product.

Chapter 6

Summary and future work

6.1 Summary

Previous sections focusing on the principles, approaches and implementation of the browser have shown the necessity for a browser to view and analyze multidimensional data sets. This thesis has presented a general purpose browser implementation and the three steps required to use the browser. First, data is collected in a multidimensional warehouse file. Second, this data is read into memory. Finally, the browser interface provides users an easy to use method to view and manipulate the multidimensional data. This browser allows users to eliminate difficulties in dealing with complex multidimensional data by isolating a two dimensional slice. The user specifies the X and Y dimensions and fixes the other dimensional coordinates easily using the interface provided by the browser. As shown in chapter 5, this browser exhibits some properties of an OLAP product. This browser is implemented for GLU program I/O.

One very important improvement would be to link the browser directly to the GLU program while it runs. This would avoid the need to down load data into a warehouse - GLU has its own warehouse. Also, it would allow us to browse infinite data structures, because the browser would request values only when they were needed for the display.

If we were able to do this, we would have an approach to the long standing problem of input/output for functional languages. A plug-in browser could make demand-

driven output practical. The browser would translate the users mouse clicks and slider movements into demands for individual data items, which would be computed as needed by the GLU program. There would be no need to call C code and invoke side effects from within GLU.

Originally, we wanted to do this, but we found out it was impractical. We would have to make changes to GLU - to allow globally declared dimensions, and to allow C functions to run concurrently with the GLU program and inject their own demands. This could be quite difficult, even if we had access to the proprietary source code. But the problems are with GLU, not our browser. The browser accesses the warehouse in a demand-driven way.

Now, four main contributions for this browser implementation have been followed in this thesis.

1. We found two approaches to navigate and collect the multidimensional output data of GLU programs.
2. We implemented and tested one of these approaches.
3. We designed a simple storage structure, a storage order and a data access address formula for a multidimensional warehouse. In fact, other data warehouses also can be imported and transformed into the multidimensional warehouse.
4. We designed and implemented a multidimensional spreadsheet-style browser to display, analyze and calculate with data in the warehouse.

6.2 Future work

Two possible directions exist for future work; one is to add enhanced functions to the current browser; the other is to navigate multidimensional positions by combining the two approaches for GLU as mentioned in section 3.2.

6.2.1 Enhancing the current browser

The current implemented version is a demonstration version which just supports some basic features, a single multidimensional warehouse, and a few functions. More work could be done on this browser. For example, this browser can be adapted for nested browsing, for multiple multidimensional warehouses connection and for consolidation.

Nested browsing means that the browser window view is not just some single cells, instead each cell can link to another browser window view or a complex structure of multidimensional data sets such as matrix, record etc. The nested browser can be used to display the complex structure of multidimensional data sets.

In this thesis, only one multidimensional warehouse is browsed by the browser. In fact, multiple multidimensional warehouses could be linked by keys. Each dimension can be a key in the multidimensional warehouse. Using linking, users can deal with larger multidimensional warehouse, more dimensions and more information. Users can link different multidimensional warehouses as they desire. Therefore, an improved browser could not only link multiple multidimensional warehouse, but can also browse a linked multidimensional warehouse.

This thesis implemented a one level consolidation path, i.e. each dimensional series of coordinates will not have a next level sub-coordinate. This browser could be improved by having unlimited coordinate levels. Based on these different levels, users can use the improved browser to display multidimensional data by roll-up or drill-down.

Moreover, Dr. Wadge has suggested a simple way to extend the way the browser handles aggregation. Currently, the browser presents extra rows and columns in the display dimension labelled with the aggregation operators - SUM, AVG, MAX and MIN. This allows the user to see the results of these operations without adding these results to the warehouse (it is very important that a data warehouse be read-only). However, it allows only one level of aggregation and in only one dimension, and this dimension has

to be one of the display dimensions.

The basic idea is that SUM and the others are treated like extra coordinates in these dimensions. Dr. Wadge suggests adding similar aggregations points to the other dimensions as well, and letting the user fix the other dimensions at aggregation points. This would mean that the user wants the particular aggregation operators applied in the given dimensions.

We can even allow more than one dimension to be fixed at an aggregation point - the corresponding operators would be composed. Since some of the operators (such as SUM and MAX) do not commute, the user would also specify an aggregation ordering of the dimensions. The two display dimensions would be the last two in this ordering.

This extension should not be too difficult to implement. It works by extending the data objects the browser handles (coordinates) rather than by extending the functionality. In fact the main extra work is to define a function which computes the value at an extended point using the function which extracts the values at standard points. This is a simple recursion.

After the addition of these features, the browser would have more of the characteristics of an OLAP product which is a powerful tool in a hot section of the software industry.

6.2.2 Different approaches to navigating output from a GLU program

In section 3.2, two approaches to implementing the browser for analyzing the output of multidimensional programs in GLU were already discussed. A possible third approach is to combine the two approaches.

The first method is directly to call a BROWSER function inside a **where** clause. Generally, this approach is good for navigating "local" dimensional positions of a GLU program. The second approach is that the main() of a C language calls the GLU program

directly to browse the output of the GLU program. Generally, this approach is good for navigating "global" dimensional positions of a GLU program.

We think to combining these two approaches to collecting the multidimensional output data of a GLU program should be further studied.

Bibliography

[AFJW95] E. A. Ashcroft, A. A. Faustini, R. Jagannathan, W. W. Wadge, *Multidimensional Programming,* Oxford University Press, 1995.

[CC94] E. F. Codd, S, B, Codd, *OLAP (On-Line Analytical Processing) with TM/1,* E. F. Codd & Associated, U,S,A. 1994.

[CCS93a] E. F. Codd, S. B. Codd, C. T. Salley, *"Beyond Decision Support,"* page 87, Computerworld, July 26, 1993.

[CCS93b] E. F. Codd, S. B. Codd, C. T. Salley, *Providing OLAP (On-line Analytical Processing) to User-Analysts: An IT Mandate*, E. F. Codd & Associated, U,S,A. 1993.

[Dej95a] E. X. Dejesus, *"Dimensions of Data", pages 139-148*, Byte, April 1995.

[Dej95b] E. X. Dejesus, *"Data to the Nth Dimension", pages 193-194*, Byte, December, 1995.

[Du86] W. C. Du, *"An Intensional 3-D Spreadsheet and its Implementation",* Department of Computer Science, University of Victoria, 1986.

[DW90a] W. C. Du, W. W. Wadge, *"A 3D Spreadsheet Based on Intensional*

Logic," IEEE Software, 7(3):78-89, May 1990.

[DW90b] W. C. Du, W. W. Wadge, *"The Eductive Implementation of a Three-dimensional Spreadsheet,"* Software-Practice and Experience, vol.20(11), 1097-1114, November 1990.

[Fin93] R. Finkelstein, *Understanding the Need for On-Line Analytical Servers,* Performance Computing, Inc., Chicago, Illinois, U,S,A. 1993.

[FJ91] A. A. Faustini, R. Jagannathan, *"Indexical Lucid," In Proceedings Fourth International Symposium on Lucid and International Programming (ISLIP 91),* SRI International, Menlo Park, California 94025, U.S.A, 1991.

[FJ93] A. A. Faustini, R. Jagannathan, *"Multidimensional Problem Solving in Lucid," Technical Report SRI-CSL-93-03,* Computer Science Laboratory, SRI International, Menlo Park, California 94025, U.S.A, 1993.

[FW87] A. A. Faustini, W. W. Wadge, *"An Eductive For The Language Plucid," ACM* 0-89791-235-7, 1987, pp86-91.

[GD94] D. Gasteiger, R. Dobson, *"Into the Enterprise,"* pages 24-25, *Byte*, September 1994.

[HCC83] T. B. Henderson, D. F. Cobb, G. B. Cobb, *Spreadsheet Software From VisiCalc to 1-2-3,* Que Corporation, 1983.

[Jag94] R. Jagannathan, *Coarse-Grain Dataflow Programming of Conventional Parallel Computers, in Advanced Topics in Dataflow Computing and Multithreading,* IEEE Computer Society, April 1995, ISBN 0-8186-6542-4.

[JD94] R. Jagannathan, C. Dodd, *GLU Programmer's Guide(version 0.9),* Technical Report SRI-CSL-94-06, Computer Science Laboratory, SRI International, Menlo Park, California 94025, U.S.A, July 1994.

[JW93] Simon L Peyton Jones Philip Wadler, *Imperative functional programming, 20'th ACM Symposium on principles of Programming Languages, Charleston, South Carolina,* January 1993.

[McM87] R. McMullan, *SuperCalc Prompt (2nd edition),* 8 Grafton Street, London W1, 1987.

[Men93] D. Menninger, *OLAP, Turning Corporate Data Into Business Intelligence,* IRI Software, U,S,A. 1993.

[Ref94] B. J. Reff, *"Dynamic-Viewing Spreadsheets,"* pages 255-258, Byte, November 1994.

[Ric94] M. Ricciuti, *"Winning The Comoetitive Game,"* pages 21-26, *Datamation,* February15, 1994.

[Sta93] M. Stadelmann, *"The Design and Implement of a Spreadsheet Based on Constraints"*, Department of Computer Science, University of Victoria, 1993.

[WA85] W. W. Wadge, E. A. Ashcroft, *Lucid, the Dataflow Programming Language,* Academic Press, 1985.

[Wat94] K. Watterson, *"The Changing World of EIS,"* pages 183-193, *Byte,* June 1994.

[Wat95] K. Watterson, *"A Data Miner's Tools,"* pages 91-96, *Byte,* October 1995.

[Wu94] Q. Wu, *"3D Structure Spreadsheet"*, Department of Computer Science, University of Victoria, 1994.

VITA

Surname: Wang Given Names: Feng

Place of Birth: _____ Date of Birth: _____

Educational Institutions Attended:

University of Victoria	1993 -- 1995
Shandong University	1982 -- 1986

Degrees Awarded:

B. Sc.	Shandong University	1986

PARTIAL COPYRIGHT LICENSE

I hereby grant the right to lend my thesis to users of the University of Victoria Library, and to make single copies only for such users or in response to a request from the Library of any other university, or similar institution, on its behalf or for one of its users. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by me or a member of the University designated by me. It is understood that copying or publication of this thesis for financial gain shall not be allow without my written permission.

Title of Thesis: A MULTIDIMENSIONAL SPREADSHEET-STYLE BROWSER

Author [redacted]

(Signature)

FENG WANG

(Name in Block Letters)

26/01/96

(Date)